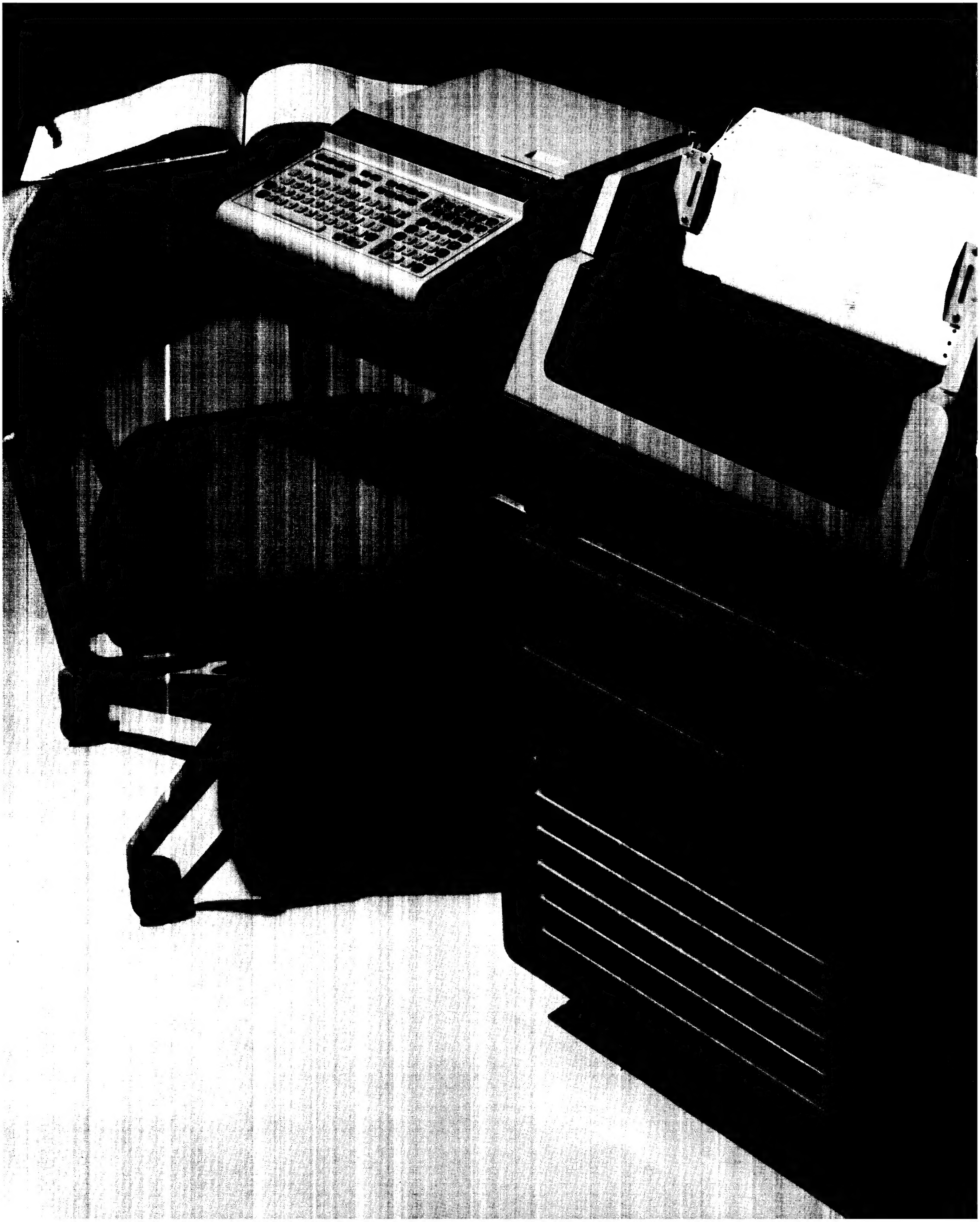
HP 9831A Desktop Computer Flexible Disk Operating & Programming

This manual describes installing and operating HP 9885M/S Flexible Disk Drives. All operations available with the 98218A Flexible Disk ROM are covered here.

The operating instructions here assume that you are familiar with programming the HP 9831A Desktop Computer, as explained in its operating and programming manual.

Hewlett-Packard Fort Collins Division
P.O. Box 1550, Fort Collins, Colorado 80521, Tel. (303)221-5000
(For World-wide Sales and Service Offices see back of manual.)



Manual Summary

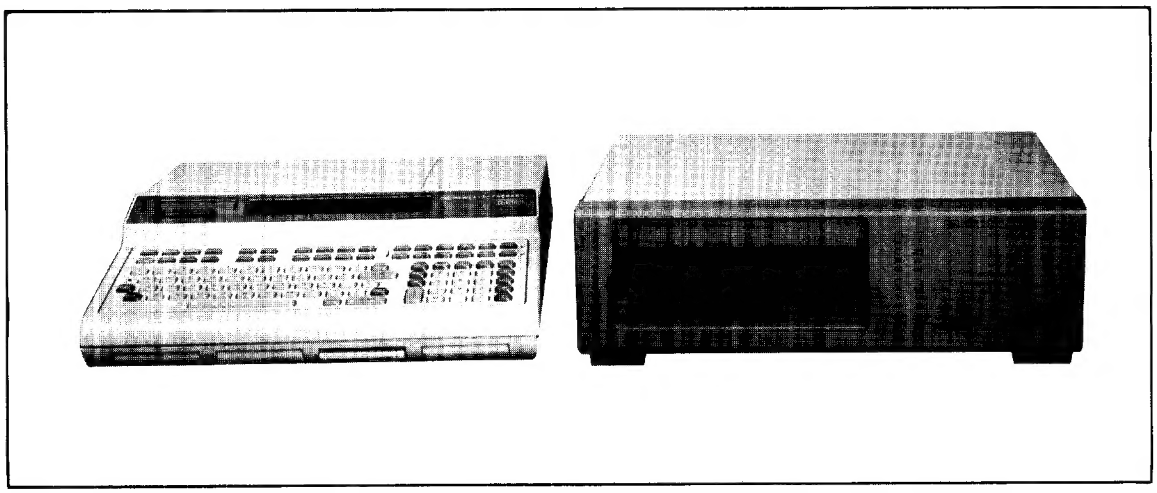
Table of Contents

Chapter 1
General Information

Introduction

The HP 9885 Disk Drive is a mass storage device that uses a flexible disk as the storage medium. Flexible disks can be accessed much faster than tape cartridges and have more than twice the storage capacity. Almost ¼ million words can be stored on each disk. The drive's short data access time, combined with disk and file access by name, make it an extremely powerful yet easy to use storage unit for the HP 9831A Desktop Computer.

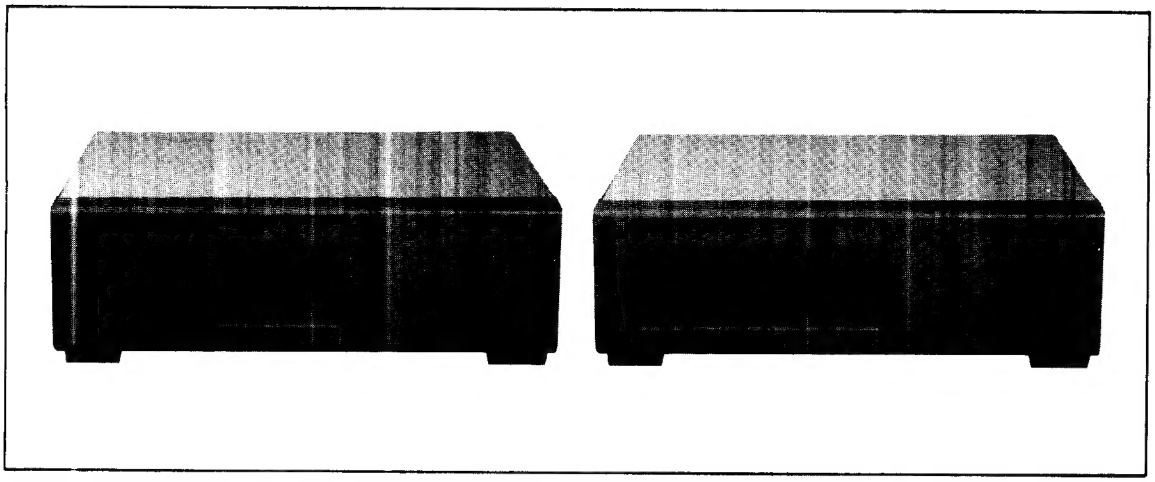
This manual shows how to install each 9885 Disk Drive and explains controlling it with your desktop computer. The disk control statements and functions are provided via the HP 98218A Flexible Disk ROM.



The 9831A and 9885 Disk Drive

The 9885M Disk Drive

The 9885M is the master drive, or the controller, in single and multiple drive systems. The 9885M can hold and operate one flexible disk at a time. At least one 9885M is required for the system to operate, although up to eight 9885M's can be connected in one system. An HP 9878A I/O Expander is required if more than three peripherals are to be connected to the desktop computer.



HP 9885M and S Flexible Disk Drives

The 9885S Disk Drive

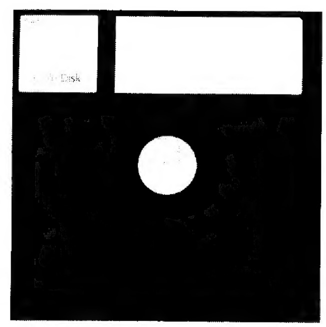
The 9885S is the slave drive used in multiple drive systems. Up to three 9885S Drives can be connected to each 9885M in the system. Each drive can hold and operate one flexible disk at a time.

The Flexible Disk ROM

The HP 98218A Flexible Disk ROM plugs into the desktop computer and adds the statements and functions for controlling 9885 Disk Drives. The ROM is supplied with a 9885M Option 031 Drive.

The Disk

The flexible disk is the storage medium for the 9885. Each disk can hold about ¼ million words. Only one side of the disk is used for storage. Be sure to read and follow the Disk Care Guidelines described in Appendix A of this manual.



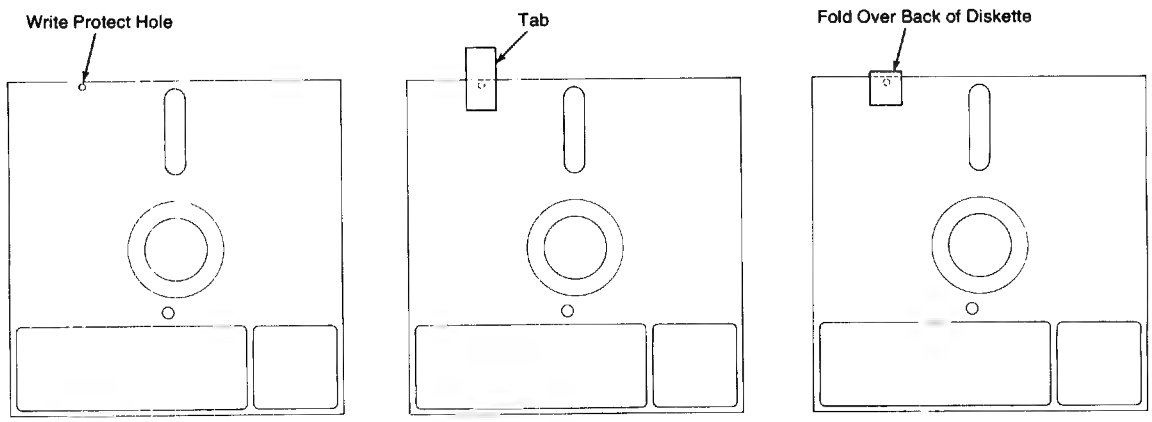
The Flexible Disk

Each disk must be initialized before it can be used for the first time. One of the disks supplied with the 9885M has been initialized and is ready for use. To initialize other disks, refer to the procedure in Appendix C.

1

Write-Protecting the Disk

The data and programs on a disk can be protected from being written over. The disk is write-protected by uncovering a hole in the sealed protective jacket at the location shown below. When the write protect hole is uncovered, nothing can be written on the disk. When the write protect hole is covered, as shown below, writing is allowed on the disk. HP disks are supplied with the hole covered, enabling you to write on the disk.



A package of opaque WRITE tabs is supplied with each disk drive. Any opaque tape, such as black electrical tape, can also be used.

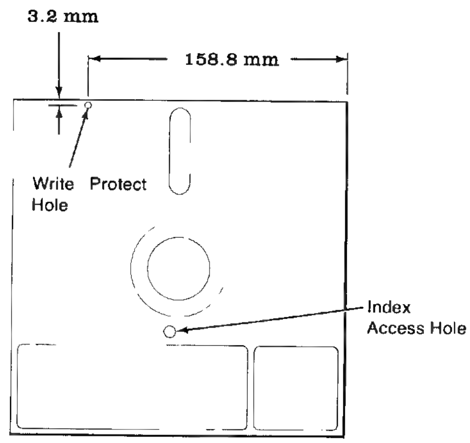
Suggested Disk Manufacturers

A list of approved disk manufacturers is available through your HP sales and service office. Use only those disks with your 9885 Flexible Disk Drive. Loss of data, damage to the read/write head, and high maintenance costs are likely to result from use of non-approved disks.

CAUTION

DO NOT USE DISKS OTHER THAN THOSE APPROVED BY HP, OTHERWISE PERMANENT DAMAGE TO YOUR DRIVE MAY RESULT.

If a disk does not have the write protect hole (see previous section), you can punch a 3.2 mm (⅛ inch) hole in the disk jacket at the location shown here –



Getting Started

An initialized disk is supplied with your system. With this disk installed, you can begin using your system immediately. Follow the steps below to get your system set up and ready to use (referring to the detailed instructions in Appendix A, when necessary).

- [ ] Unpack your 9831A and 9885M Drive(s) and check them for physical damage; complete unpacking instructions are on page 73.

- [ ] Check for the appropriate fuse, line voltage and power cords; more electrical information is on page 75.

WARNING

ALWAYS DISCONNECT THE DRIVE FROM AC POWER BEFORE CHANGING FUSES OR SETTING VOLTAGE SELECTOR SWITCHES.

- [ ] Connect the desktop computer to the drive(s) using the interface cable(s) supplied. Then connect the system to an ac power source; for further information, see page 77.

Set the drive number on the back panel of each drive; the drive number (0 thru 3) selected is the one opposite the dot on the switch. Then set the select code switch (8 thru 15) on each interface card used; see page 78 for additional information.

- [ ] Install the Flexible Disk ROM in the desktop computer; instructions are on page 79.

- [ ] Install the HP disk with the "Initialized" label if you want to use your system immediately. (The other disk provided is not initialized.) Instructions for installing a disk are on page 79 .

- [ ] Switch the desktop computer and disk drive(s) on.

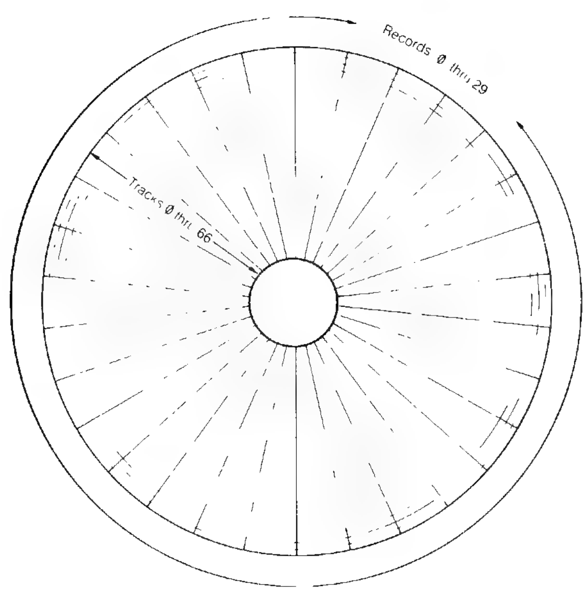
- [ ] To be sure your system is installed and functioning properly, perform the 9885 tests in the System Test Booklet supplied with the 9831A. Before continuing to the next chapters on disk operations, be sure to read the followng pages on Disk Structure, File Structure and Syntax Guidelines.

- [ ] To initialize a blank disk, follow the procedure beginning on page 96 .

Disk Structure

The disk used in the 9885 is a circle of plastic 20 cm (7⅞ inches) in diameter, enclosed in a sealed black plastic jacket. Bonded onto the surface of the disk is a ferromagnetic iron oxide with characteristics similar to magnetic tape. Data is stored in the form of binary digits represented by magnetized spots on the disk. Information is stored and retrieved by a read/write head that comes in contact with the lower surface of the disk.

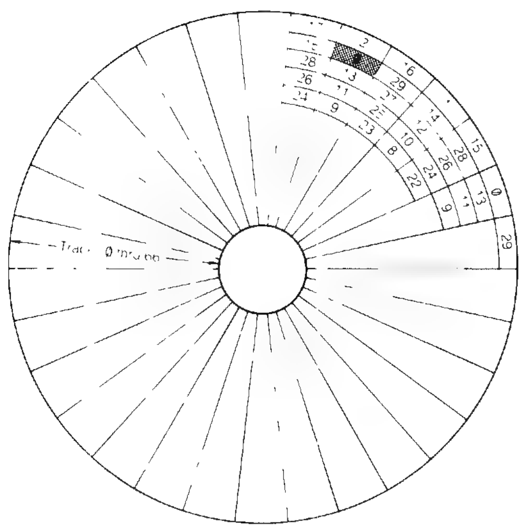
Data is stored in concentric tracks on the disk. Each disk has 67 tracks, numbered 0 thru 66. The disk is also subdivided into 30 pie-shaped sections called records (1 record = 128 words).



Disk Structure

Records are not numbered sequentially; they are numbered alternately, instead. This shortens the time it takes to access a record since a complete revolution of the disk between execution of read (or write) statements is avoided.

A diagram of disk tracks and records with their alternating numbering system is shown next. The shaded area shows the location of a specific record – Track 1, Record 26.



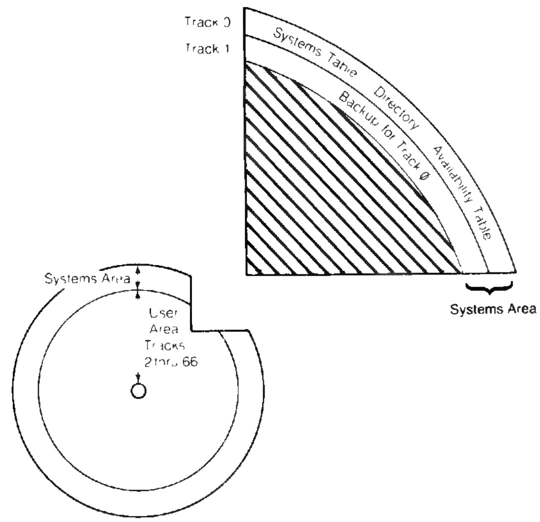
Flexible Disk Records

In addition to an alternating numbering system, the location of the beginning record (record 0) of each track is skewed to avoid a revolution when the drive accesses a new track. For example, after Record 29, Track 0 is accessed, then Record 0, Track 1 is accessed without an extra revolution.

Systems Area

Some of the area on the disk is reserved for use by the system (tracks 0 and 1). The rest of the disk area (tracks 2 thru 66) is available for your use. Each track in the systems area has a duplicate copy of these items –

- Systems Table.
- Directory to file locations and their sizes (once you've defined them).
- Availability Table that indicates remaining usable disk space.



Systems Area of Disk

Systems Table

Record 0 of the systems table indicates the computer (e.g., an HP 9831A or 9825A) used to initialize the disk, an optional disk label (name), the number of defective tracks, and the location of the beginning of the user area.

When a disk is initialized, the number of defective tracks is recorded in the systems table. If more than six tracks are defective, the disk is rejected (contact HP for a replacement). The first word of the system table indicates the number of defective tracks. The physical location of the defective tracks is not accessible. This results in a contiguous set of logical tracks with no intervening defective tracks. For example, if there are two defective tracks on a disk, the usable tracks will be numbered 0 thru 64.

File Directory

The directory in records 1 thru 22 contains entries for 352 possible files, one entry for each file written on the disk. Each entry (8 words) contains information such as file name, location, size and type of each file. If the Directory in track 0 cannot be read, the spare directory from track 1 is automatically read.

1

Availability Table

The availability table in records 23 thru 28 monitors the amount and location of remaining disk space. The availability table is automatically updated after any file is added to, or removed from, the disk.

Any space on the disk that becomes available (after execution of a KILL statement) is automatically combined with other available disk space if the areas are contiguous. This creates larger available spaces on the disk instead of numerous shorter spaces.

Record 29 of the Systems Area is unused.

Backup Track

Track 1 contains the same system information as track 0. The information on track 1 is automatically used if track 0 should become defective.

Storage Area

Tracks 2 thru 66 are used for recording your files and programs. The tables in the systems area are updated whenever new information is added to, or deleted from, the disk and whenever the disk is reorganized (repacked). With 30 records per track and 128 words per record, there are 249,600 words of available storage space per disk.

File Structure

The flexible disk system is organized around user defined memory areas called **files**. Each disk can have up to 352 files, depending on the size of each file. Files can be used to hold data (data files), programs (program files), entire machine memory (memory files), special function keys (key files) and binary programs (binary files).

You create these files, name them and – for data files – specify their size. The programming statements and functions described in the next chapters enable you to store information on, and retrieve information from, your disk files.

Each file contains one or more **records**. A record contains 128 words of memory. A **word** is the smallest addressable unit of data on the disk which can be accessed directly. A file cannot be greater than 1950 records (the maximum available storage space on the disk).

The size of a program, memory or key file is automatically determined: it is the number of words (and the number of records) required to store the program. When you create a data file, however, you must specify its size in records. The differences between program files and data files follows.

Program Files

Programs are stored on a disk using as many complete records as necessary, each record containing 128 words. So if a program is 129 words long, two records are required to store it on the disk.

Data Files

There are three ways to store and access data: **serially**, **randomly**, and **logically**. It's up to you to determine which method of data access best suits your needs. Since your decision will be based on the amount of available disk storage and the time required for your operations, an understanding of data file structure is necessary for the most efficient use of your system.

For example, suppose you are working with thousands of customer account numbers and their balances due; your job is to output a daily list of all customers and their balances. In this situation, it's best to pack all data items (customer numbers and balances due) together tightly in a data file to save space on the disk and to save time when accessing the data. This is serial file access.

To update individual customer balances, you'll need another file containing customer numbers, names, addresses, items purchased and balances due. The data in this file is arranged so that each individual item (customer name or number) can be accessed. This method of storing data usually takes more space on the disk. The advantage to this method is that any item can be easily updated since individual items can be accessed much faster. This is random file access.

When you wish to update many individual portions of a file as fast as possible, logical file access can be used. Using this method allows better storage efficiency than random file access.

1

Serial File Access

Data treated as a unit of information (instead of as individual items) can be handled using serial PRINT# and serial READ# statements. When serial PRINT# statements are used to store data on the disk, data items are stored compactly without identifiable marks between items. These data items make up a file and can contain as many records as necessary. Data lists can contain numerics and strings.

All or part of the information stored originally can be retrieved in one serial READ# statement. The list of data elements read does not have to be identical to the list originally printed in the file, but these data lists must be identical in size, type and order. (The names you assign to these elements can still vary.)

The beginning of a serial file is the only point where direct access is possible. Storage space is utilized with maximum efficiency when a serial PRINT# is done, since data is packed solidly and no unused space is left between items.

Random File Access

When data items are to be handled individually (instead of as a unit), random PRINT# and random READ# operations can be used (the same PRINT# and READ# statements are used with an additional parameter to specify record number). Each data item is stored in one (or more, if required) record so that every data item is directly accessible. Storing data randomly may not utilize storage space effectively, since only a part of a record (or records) required for storage may be used.

Each of the data items stored originally can be retrieved by using a random READ#. The list of data items does not have to be identical to the list originally printed in the record, but the data items must be identical in size, type and order. (The names you assign to these elements can still vary.)

When working with data using random PRINT# and READ# statements, you specify which record within a file you want to access. The advantage of this method is that every record is directly accessible, in any order.

Logical File Access

When you wish to handle data as individual units, and also wish to specify the exact point within a record where the data is to be printed or read from, use logical file access. This access method is specified by adding another parameter, called a word pointer, to the READ# and PRINT# statements.

Logical file access offers the best accessibility to data, since you specify the exact word at which the read or print begins. Use of disk storage space is good, too, since end of record (EOR) marks are not added after the data to fill the record. So any remaining space in the record can be used for more data storage.

Comparing Data Access Methods

As mentioned before, you decide on which method of data accessing is best for your particular needs. This decision is usually not made easily, because of the advantages and disadvantages of each method. For example, more efficient storage space utilization must be sacrificed for a shorter access time, and vice versa. Once your decision has been made, it is difficult to change later, so make your decision carefully.

The advantages and disadvantages of accessing data with each method are summarized below.

Comparison of Data Access Methods

	Access Time	Storage Efficiency
Serial	Varies – longer for higher-numbered records	Best – data is packed solidly
Random	Good – direct access to any record	Varies – EOR marks fill remainder of each record
Logical	Best – only part of a record need be accessed	Good – EORs not used

Syntax Guidelines

The disk operations available with the Flexible Disk ROM are described in the next chapters. The conventions and terms most often used within the syntaxes for statements and functions are listed here.

brackets [ ]	Items enclosed in brackets are optional.
`dot matrix`	Items in dot matrix must appear as shown.
...	Dots tell you that the preceding item can be repeated.
drive no.	An integer expression from 1 thru 3 indicating which drive should be used. Also see "The UNIT Statement" in Chapter 4.
file name	The name used to define a specific file. It can contain up to six characters; quotes (`"`), commas, colons, blank spaces, a leading asterisk (`*`), and nulls (decimal 0) cannot be used in the file name. The name can be either text (characters within quotes) or a string variable (quotes not used).
1st line number	An integer number referencing a program line.
2nd line number	The 2nd line number can be used only with the 1st line number.
file no.	An integer expression from 1 thru 10 representing a file name, as specified via a FILES or ASSIGN statement.
record no.	An integer expression specifying a physical record within a file.
word pointer	An integer expression from 1 thru 129 specifying the starting point (word) for logical PRINT# and READ# operations.

An "integer expression" can be an integer number (like `2`), a variable (like `A` or `C3`), or an expression (like `A+2`). Non-integer values are rounded before being used.

All disk statements and functions can be executed either from the keyboard or a program. A complete list of disk BASIC syntax is in Appendix D and in the BASIC Reference Booklet supplied with the desktop computer.

Chapter 2

Program File Operations

Introduction

Disk files can be used to hold data (data files), programs (program or binary files), special function keys (key files), or the entire read/write memory (memory files). This chapter introduces program, memory, binary, and key files and the statements used to manipulate them. It is assumed here that you are working with one disk drive. For information on operating more than one drive at a time, see The UNIT Statement in Chapter 5.

The statements most often used when working with program files are –

SAVE RESAVE	Store program lines into a disk file.
CAT	Lists information about each file on the disk.
GET	Loads program lines from the disk to the memory.
CHAIN	Same as GET, except that program variables are not erased.
KILL	Erases a file name from the disk.
SAVE KEY GET KEY	Store and load information on special function keys.
SAVE MEM GET MEM	Store and load the entire read/write memory.
GET BIN	Load binary information from a prerecorded disk.

The SAVE Statement

SAVE file name [, 1st line number [, 2nd line number]]

The SAVE statement initializes a file name and copies all or part of the current program into the file. The operation of SAVE parallels the STORE (tape cartridge) statement. The size of the file is automatically set by the size of the program copied onto the disk.

2

The file name must be unique. If you try to save a program using a name that is already in use on the same disk, the SAVE is cancelled and ERROR 92 is displayed.

The optional line number parameters enable you to save part of your program, rather than all of it. With one line number specified, SAVE stores all lines after and including the specified line. With both parameters, the lines between and including the specified lines are stored onto the disk.

For example, here's a program that inputs values and computes their average (mean) value. Then if the operator wishes, the input values can be printed.

```
10 DIM A[500],A$[10]
20 A=0
30 FOR I=1 TO 500
40 DISP "ENTER VALUE";
50 INPUT V
60 IF V<0 THEN 100
70 A[N]=V
80 A=A+V
90 NEXT N
100 PRINT "Average of "N"values equals"A/N
110 DISP "PRINT ALL VALUES (yes or no)";
120 INPUT A$
130 IF A$="yes" THEN 150
140 END
150 SERROR E,210
160 PRINT LIN2
170 FOR I=1 TO 500 STEP 5
180 PRINT A[I],A[I+1],A[I+2],A[I+3],A[I+4]
190 NEXT I
200 END
210 IF E=40 THEN 240
220 DISP "ERROR"E"IN PRINTOUT ROUTINE";
230 END
240 PRINT LIN5
250 END
```

Once the program is in the desktop computer, you can save the entire program in a file named "master", for example, by executing –

```
SAVE "master"
```

You could also save portions of the program. For example, to save lines 150 thru 250 in a file named "print", execute –

```
SAVE "print",150
```


Or to save lines 10 thru 140 in a file named "avrge" (for average), execute –

```
SAVE "avrge",10,140
```

Once a file has been set up using SAVE, it cannot be overwritten by using another SAVE. So, after editing a program which has already been saved, use the RESAVE statement to save the new program in the same file.

The RESAVE Statement

RESAVE file name [, 1st line number [, 2nd line number]]

The RESAVE statement allows you to copy the current program lines into a file which already exists on the disk. RESAVE automatically adjusts the file's size to hold the new program; all lines previously in the file are erased.

The CAT Statement

CAT [drive no.[, printer select code]]

The CAT (catalog) statement lists information about each file on the disk. When the printer select code is not specified, the standard printer is used. When the drive number is not specified, the drive specified by a UNIT statement (or drive no. 0 on select code 8) is used.

Here's an example printout –

```
          CATALOG OF DRIVE  0
        AVAILABLE RECORDS:  1205
NAME     TYPE    LENGTH   TRACK   RECORD
----     ----    ------   -----   ------

keeper   OTHER    100R     10       50
calc     PROG    4798W      6       10
flist    PROG    1542W     19        6
lister   PROG     909W      7       18
CBtemp   DATA     250R      8        1
SORTIE   PROG     106W     18       17
calc2    PROG    4597W     16       11
↑LIST    DATA      30R     17       17
binary   BINARY    76W      7       28
keys     KEYS       4W      7       26
t        PROG      80W      7       27
dtest    PROG     246W     19       19
tester   PROG    2556W     19       21
disct    BINARY  1024W     20       11
bin      MEMORY  8165W     20       19
dup      MEMORY  8165W     22       23
XXX      BINARY   319W     26       22
```


Notice that the file size (length) is given in words for program, key, memory and binary files. All other types of files are listed in records. Also notice that TRACK and RECORD designate the location of the particular file on your disk. Since the order in which the files were created may differ, these numbers vary from disk to disk and drive to drive. The file type OTHER indicates a file generated by an HP computer other than the 9831A.

The GET Statement

GET file name [, 1st line number[, 2nd line number]]

The GET statement loads a program from the disk to the 9831A. The values of all variables not defined in a COM statement become undefined. GET can also renumber the statements of the program and run it without further instructions. This statement parallels operation of the LOAD (tape cartridge) statement.

When GET is executed, all program lines in the specified file are loaded into memory. All program lines previously in the memory are erased unless the 1st line number is specified.

When the 1st line number is specified, the loaded program lines are renumbered with the beginning line number corresponding to the specified 1st line number. Program lines previously in memory, with line numbers lower than the 1st line number, are retained; other lines previously in memory are erased.

After GET is executed from the keyboard –

- If the 2nd line number is not specified, the machine halts.
- If the 2nd line number is specified, program execution begins at that line number.

After GET is executed from a program –

- If the 2nd line number is not specified, program execution is restarted either with the program line immediately following GET in the original program, or with the first line of the loaded program (if there were no lines after the GET statement in the original program, or if the lines were erased by GET).
- If the 2nd line number is specified, program execution is restarted with that line number.

For example, to load the program previously stored in the file named "master" (see page 15), execute –

```
GET "master"
```

If you wish to load the same program again, but not erase the program in memory, execute –

```
GET "master";300
```

As shown in the following listing, the program originally loaded has not been erased; the second program, beginning on line 300, has been loaded after it. Had the second program been renumbered from line 30, only lines 10 and 20 of the original program would have remained; all lines from 30 on would have been either printed over or erased.

2

```
10 DIM A[500],A$[10]
20 A=0
30 FOR I=1 TO 500
40 DISP "ENTER VALUE";
50 INPUT V
60 IF V<0 THEN 100
70 A[N]=V
80 A=A+V
90 NEXT N
100 PRINT "Average of "N"values equals"A/N
110 DISP "PRINT ALL VALUES (yes or no)";
120 INPUT A$
130 IF A$="yes" THEN 150
140 END
150 SERROR E,210
160 PRINT LIN2
170 FOR I=1 TO 500 STEP 5
180 PRINT A[I],A[I+1],A[I+2],A[I+3],A[I+4]
190 NEXT I
200 END
210 IF E=40 THEN 240
220 DISP "ERROR"E"IN PRINTOUT ROUTINE";
230 END
240 PRINT LIN5
250 END
```

first program

```
300 DIM A[500],A$[10]
310 A=0
320 FOR I=1 TO 500
330 DISP "ENTER VALUE";
340 INPUT V
350 IF V<0 THEN 390
360 A[N]=V
370 A=A+V
380 NEXT N
390 PRINT "Average of "N"values equals"A/N
400 DISP "PRINT ALL VALUES (yes or no)";
410 INPUT A$
420 IF A$="yes" THEN 440
430 END
440 SERROR E,500
450 PRINT LIN2
460 FOR I=1 TO 500 STEP 5
470 PRINT A[I],A[I+1],A[I+2],A[I+3],A[I+4]
480 NEXT I
490 END
500 IF E=40 THEN 530
510 DISP "ERROR"E"IN PRINTOUT ROUTINE";
520 END
530 PRINT LIN5
540 END
```

second program

The GET command can also be used to begin program execution automatically. For example, to load only the printout routine previously saved in file "print" (see page 15), execute –

```
GET "print";10;1
```

The above statement says to begin running the program at line 1. Since there is no line 1, the program begins at the first available line, line 10.

2

The CHAIN Statement

CHAIN file name [; 1st line number[; 2nd line number]]

The CHAIN statement is identical to GET, except that current values of variables are not erased. This statement operates like the LINK (tape cartridge) statement.

The CHAIN statement is most often used in a program to link a program previously stored on the disk to one currently in the memory. For example, key in and save these three programs. First, to save this program –

```
10 PRINT "BEGIN executed."LIN2
20 CHAIN "MIDDLE",10,10
30 END
```

execute –

```
SAVE "BEGIN"
```

Then, to save this program –

```
10 X=0
20 GOTO 40
30 X=2
40 IF X=1 THEN 80
50 IF X=2 THEN 100
60 PRINT "MIDDLE chained to BEGIN and ran from line 10."LIN2
70 CHAIN "END",10,10
80 PRINT "MIDDLE chained to END and ran from line 40."LIN2
90 CHAIN "END",110,30
100 PRINT "END chained to MIDDLE and ran from line 30."LIN2
110 END
```

execute –

```
SAVE "MIDDLE"
```

Finally, to save this program –

```
10 IF X=2 THEN 50
20 PRINT "END chained to middle and ran from line 10."LIN2
30 X=1
40 CHAIN "MIDDLE",10,40
50 PRINT "END"LIN5
60 END
```

execute –

```
SAVE "END"
```

In the above program segments, the values of J are retained when chaining from program to program. In this way, selected portions of the last two program segments are run. These programs are run by executing –

```
GET "BEGIN",10,10
```

Here's the printout –

```
BEGIN executed.

MIDDLE chained to BEGIN and ran from line 10.

END chained to MIDDLE and ran from line 10.

MIDDLE chained to END and ran from line 40.

END chained to MIDDLE and ran from line 30.

END
```

Be careful when renumbering a program by using CHAIN or GET. It is possible that, when renumbering a program from either the keyboard or another program, a line number parameter will fall outside the range of from 1 thru 9999, causing ERROR 4.

The KILL Statement

KILL file name

The KILL statement erases the named file from the disk and releases the space it occupied for further storage. For example, to erase the file named "print" recorded in a previous example, execute –

```
KILL "print"
```

An automatic update of the availability table is performed following the KILL statement. This means that the space released is automatically combined with any contiguous space on the disk to create larger available storage areas. For example in the diagram below, files 0, 2, and 4 were previously killed.

file 0	file 1	file 2	file 3	file 4

If file 1 is killed, the contiguous areas held by files 0, 1, and 2 are combined and becomes one larger available area instead of three smaller available areas. Notice that the area of file 4, however, is not combined.

The SAVE KEY Statement

SAVE KEY file name

The SAVE KEY statement stores the current special function key definitions on the specified file. SAVE KEY operates like the STORE KEY (tape cartridge) statement.

The definitions of all 24 special function keys can be stored in one file at a time. Since only key definitions are saved with this statement, two files are required to save a program which uses special function keys: one to save the program and one to save the special function key definitions.

The GET KEY Statement

`GET KEY` file name

The GET KEY statement loads special function key definitions from the specified file of the disk to the special function keys. The keys are then redefined as when they were before the information was saved using SAVE KEY. The GET KEY statement works like the LOAD KEY (tape cartridge) statement.

2

All program variables are erased by GET KEY.

The SAVE MEM Statement

`SAVE MEM` file name

The SAVE MEM (save memory) statement copies the entire read/write memory onto a single file, and is similar in operation to the STORE MEM (tape cartridge) statement. All program lines, variables, operating modes, subroutine pointers, etc. are copied with SAVE MEM, allowing you to suspend 9831A status on disk. Use GET MEM to reload the 9831A at a later time.

The GET MEM Statement

`GET MEM` file name

The GET MEM (get memory) statement loads a memory file previously stored with SAVE MEM. The entire read/write memory is loaded using GET MEM; see the previous section.

The GET BIN Statement

`GET BIN` file name

The GET BIN (get binary) statement loads binary-coded information from a prerecorded disk file. GET BIN works like the LOAD BIN (tape cartridge) statement. Remember that binary information cannot be viewed, listed, or rerecorded on tape or disk.

Chapter 3
Data File Operations

Introduction

This chapter covers the statements and functions used when working with data files. If you have just read the previous chapter, you'll notice that the CAT (catalog) and KILL statements are used in the same way here as with program files.

As in the previous chapter, it's assumed here that you are working with one disk drive. For information on controlling more than one drive, see The UNIT Statement in Chapter 5.

The data file statements are –

OPEN	Sets up and names each data file.
CAT	Lists information about each file on the disk.
KILL	Erases a specified file.
FILES	Indicates which files should be used in subsequent disk operations.
ASSIGN	Assigns data files names and determines the status of an existing file.
PRINT#	Prints data into a specified file.
READ#	Reads data from a specified file.
IF END#	Automatically exits a READ# or PRINT# operation when an end of file (EOF) mark is seen.

These functions are described here –

AVAIL	Returns the total number of records available (unused) on the disk.
FIL	Returns the size of the largest unused area on the disk.
TYP	Determines the type of the next data item to be read.
SLEN	If the next item to be read is a string variable, the string length is returned. When the next data item is not a string, −1 is returned.
SIZE	Returns the size of a specified file.
REC	Returns the current position of the record pointer in a specified file.
WRD	Returns the current position of the word pointer for a specified file; use with logical file access.

3

Overview of Data File Operations

Before you can do any data file operations, you must first find a free area on the disk and give that area a name. This is done with the OPEN statement, which defines that area as a data file. The file can be used to contain whatever data you want to put into it, like readings from an instrument, inventory information, etc.

Since there can be numerous data files on a disk, a program must indicate which file(s) are to be used at any given time. This is done by including a FILES or ASSIGN statement in the program.

Once the program indicates the data files to be used, it can store information (numbers and strings) in the files using PRINT# statements and can read back the information using READ# statements. The parameters used in each PRINT# and READ# determine the data access method to be used: serial, random, or logical. Each method is explained in the following pages.

The IF END# statement and the many functions described in this chapter expand your control of data file operations.

The OPEN Statement

OPEN file name , number of records

The OPEN statement creates a data file with a specified number of physical records, assigns it a name, and places an end of file (EOF) mark in each word of each record in the file. An OPEN statement indicating file name and size must be executed before data can be printed in that data file. Each data file must be assigned a unique name. The number of records parameter can be an integer or an expression. A file can contain from one record (128 words) to 1,950 records. Attempting to open a file larger than 1,950 records results in ERROR 95. That error also indicates an attempt to open a file larger than the largest free space currently available.

3

The first statement in many of the following example programs illustrating data storage and retrieval is an OPEN statement. The OPEN statement is included only to remind you that data files must be opened before data can be printed in them. It's best to execute the OPEN statement from the keyboard, however, since ERROR 92 results when you run the same program (open the same file) more than once.

The CAT statement and the AVAIL and FIL functions can be used to help you open files.

The CAT Statement

CAT

The CAT (catalog) statement lists information about each file on the disk currently addressed. See page 16 for a sample printout and further information.

The AVAIL Function

AVAIL drive no.

The AVAIL (available records) function returns the total number of records available (unused) on the specified disk. This is the same number printed at the start of each CAT listing.

The FIL Function

FIL drive no.

The FIL (file) function returns the size (in records) of the largest unused space available on the specified disk. This number can then be used to open the largest possible data file. For example, this statement –

```
OPEN "large";FIL 0
```

opens a file named "large"; it's size is determined by the largest unused area on the disk in drive number 0.

3

The KILL Statement

KILL file name

The KILL statement erases the named file from the disk and releases the space it occupied for further storage. For more information, see page 21.

The FILES Statement

FILES file name$_1$ or * [# drive no.$_1$][, file name$_2$ or * [# drive no.$_2$]...]

The FILES statement indicates which data files are to be used, and optionally, the number of the drive accessed for each file name. The files listed are assigned numbers in the order in which they appear in the FILES list. These file numbers are then used to reference specific files in PRINT# and READ# statements and various functions described later.

For example, data A is assigned file number 1 and data B is assigned file number 2 in the following statement –

```
10 FILES dataA,dataB
```

Up to ten file names, each with an optional drive number, can be specified in each statement. As in the example, the position of each file name in the list determines its file number. Notice that you do not use quotation marks around each file name. Also, string variables cannot be used in FILES statements.

A single asterisk (*) can be used in place of a file name; this indicates that you wish to specify the name of a file in a later ASSIGN statement, as explained in the next section.

The optional drive number can be an integer from 0 thru 3; it specifies the drive to be used for that file, allowing you to specify more than one drive from a FILES statement. When a drive no. is not specified, the drive specified by the last UNIT statement is automatically used. (Drive no. 0 is used when UNIT is not given.) The UNIT statement is described in Chapter 5.

3

Each new FILES statement cancels any previous FILES list. To cancel the last FILES list, execute `FILES *`. The FILES list is also erased by executing `ERASE A` or `LOAD BIN`, or by switching the desktop computer off.

Data Pointers

A **record pointer** for each file in the FILE list is automatically maintained. This pointer is used to specify at which record data storage or retrieval begins in the file. A **word pointer** is also maintained for each current record; it points to the first word of the next data item to be accessed in the record.

After executing a FILES (or ASSIGN) statement, the record pointer is positioned at the beginning of the first record in a file. The word pointer is then incremented through the record as data items are stored (PRINT#) or retrieved (READ#). A new FILES statement obsoletes the previous one and also resets all pointers for the specified files. Executing a UNIT statement without a select code does not affect data pointers. When a select code is used, however, all pointers may be reset; see The UNIT Statement in Chapter 5. All pointers are reset for a specific drive when the door to that drive is opened.

The current position of each pointer can be found by using the REC (record) and WRD (word) functions, as described at the end of the chapter.

The ASSIGN Statement

ASSIGN file name ; file no. [; return variable[; drive no.]]

The ASSIGN statement assigns a file number to any previously opened file name. The file number can be –

- a number previously reserved in a FILES list with an *.
- an unused file number (to assign a new number).
- a number previously used (to re-assign an old number).

3

The optional return variable is used to determine a file's status. This parameter can be either a simple or an array variable.

Since a FILES statement can contain up to ten file names or asterisks, the file number in an ASSIGN statement must be a positive integer from 1 thru 10. For example –

```
10 FILES data1,*,data3,*
20 ASSIGN "data2",2,V
```

In this example, the first asterisk in the FILES statement is assigned the file name "data2". (Assume that the data file, data2, was opened before the ASSIGN statement was executed.) Additional ASSIGN statements can be placed later in the same program to reassign a different file name to any file number. Each ASSIGN statement sets the data pointers to the first item of the first record in the specified file.

V is the return variable in the previous example. Its value is assigned during execution of the ASSIGN statement and can then be used anytime in the program. The value of the return variable indicates these conditions –

Return Value	Meaning
0	File is available.
1	File type is not data.
2	Drive no. is not from 0 thru 3.
3	File has not been opened.
4	File no. is not from 1 thru 10.

By checking the value of the return variable, you can avoid errors like ERROR 90 (file not found).

The following program shows how the ASSIGN statement can be used with a string variable to open new data files.

```
10 DISP "OPEN NEW DATA FILES"
20 WAIT 2000
30 DIM A$[6]
40 DISP "FILE NAME";
50 INPUT A$
60 ASSIGN A$,1,V
70 IF V=3 THEN 110
80 DISP "FILE NAME "A$" ALREADY EXISTS!"
90 WAIT 1000
100 GOTO 40
110 DISP "NUMBER OF RECORDS";
120 INPUT R
130 OPEN A$,R
140 DISP A$" OPENED WITH"R"RECORDS.";
150 WAIT 1000
160 GOTO 40
170 END
```

In this example, line 70 branches to line 110 if the file name you enter does not exist, and opens the file. Without this IF statement, ERROR 92 occurs when you attempt to create a file which has been previously opened. Use of the return variable avoids that error message.

Remember that a string variable cannot be used directly as a file name in a FILES statement, but a string variable can be used in an ASSIGN statement, as shown in the previous program.

As shown in the example, it's not necessary to use a FILES statement before using ASSIGN. The ASSIGN statement can be used to set the data pointer to the first item of a specified file without affecting file pointers for any other files previously specified.

ASSIGN can also be used to reassign file positons (numbers) from the previous FILES statement. For example, in the sequence on the next page –

```
10 FILES names,IDs,*,grades
20 ASSIGN "class",3,V
 •
 •
 •
100 ASSIGN "report",4,V
 •
 •
 •
500 ASSIGN "finals",5,V,1
```

Line 20 assigns the file named "class" to the third position (number 3) in the FILES list. Then line 100 reassigns the fourth position to be a file named "report". Finally, line 500 adds a fifth position, a file named "finals". Notice that finals is on drive no. 1.

Data Access Methods

The PRINT# and READ# statements allow you to store and retrieve data in any of three methods: serial, random or logical. Briefly stated, serial access handles data in blocks, random access handles data in records, and logical access handles data in item-by-item quantities. Each method offers unique advantages, as explained in Chapter 1.

You specify which data access method to use by including the appropriate parameters in each PRINT# and READ# statement. The syntaxes for each statement and method are shown in the following sections. The IF END# statement and various data file functions are described at the end of the chapter.

Serial File Access

Serial file access is used to handle data in blocks. As shown in the READ# and PRINT# syntaxes, you merely specify which file number to access – the system automatically decides where to place the data within the file.

The Serial PRINT# Statement

PRINT # file no. ; data list [; END]

The serial PRINT# statement stores data in the specific file, either after the last item read or printed, or at the beginning of the file. The list of data items can consist of constants, variables, or text. The length of the data list is limited by the length of the BASIC statement (80 characters), or by the size of the file.

After a serial PRINT# is executed, end of record (EOR) marks are used to fill the rest of the last record printed. When the optional END parameter is used, however, an end of file (EOF) mark is placed immediately after the last data item; then the rest of the record is filled with EORs. The EOF mark can be used later to find out how much data is in the file.

The record and word pointers mentioned earlier move through the file as you store or retrieve data items. Data is printed or read consecutively from the position of the pointers, which are set at the beginning of the file when a FILES or ASSIGN statement for that file is executed.

3

Here is an example using the serial PRINT# statement to record five students' identification numbers and test grades.

```
10 OPEN "ID#",1
20 OPEN "grades",1
30 FILES ID#,grades
40 FOR J=1 TO 5
50 DISP "STUDENT'S ID#";
60 INPUT I
70 PRINT #1;I
80 DISP "NEXT TEST SCORE";
90 INPUT S
100 PRINT #2;S
110 NEXT J
120 END
```

The program can be used to print these identification numbers in a file named "ID#", and the corresponding grades in a file name "grades" –

ID#	Grade
1111	88
2222	67
3333	98
4444	81
5555	99

The above program uses two separate files: one for the students' identification numbers and one for their grades. The information can be combined into one file, as in the next program.

```
10 OPEN "scores",1
20 FILES scores
30 FOR J=1 TO 5
40 DISP "STUDENT´S ID#, GRADE";
50 INPUT I,G
60 PRINT #1;I,G
70 NEXT J
80 PRINT #1;END
90 END
```

Line 60 prints the I.D. and test scores of the students into the file named "scores". The data items (I.D. numbers and grades) are printed alternately. Line 80 places an EOF mark after the five sets of data are printed. Since an EOF mark prevents reading data beyond its position, the END parameter should be used with care.

3

Using string variables allows you to enter students' names, rather than their I.D. numbers. For instance, a string variable can replace the variable I in the last program, allowing names to be used –

Name	Grade
Suzie Page	99
Blue Gill	90
Carol Rose	88
Jack Allison	74
Sean Thomas	80

Here's a program which uses a string variable for students' names; it prints names and test scores in a file called "class" –

```
10 OPEN "class",1
20 DIM N$[15]
30 FILES class
40 FOR I=1 TO 5
50 DISP "STUDENT´S NAME";
60 INPUT N$
70 DISP "NEXT TEST SCORE";
80 INPUT S
90 PRINT #1;N$,S
100 NEXT I
110 PRINT #1;END
120 END
```

The Serial READ# Statement

READ # file no. [; data list] or [, record no.]

The serial READ# statement reads numbers and strings into variables serially from the specified file, starting after the last item printed or read. The data list can be replaced by a record number to reposition the record pointer, as explained later.

Before you can work with data which has been printed in a file, you must first read the data into the desktop computer. Remember that you are not erasing the data on the disk by reading it; data is merely copied into the variables specified. (This data can be updated and reprinted, either into the original file, without using a KILL command, or into a new file.)

For example, the program on page 31 printed data into files named "ID#" and "grades". To read the data from those files and output the data on the standard printer, use this program –

```
10 FILES ID#,grades
20 PRINT "Student ID#    Grade"
30 READ #1;I
40 READ #2;G
50 PRINT I,G
60 GOTO 30
70 END
```

The FILES statement serves two purposes in this program: it references the file number parameters in the serial READ# statements (lines 30 and 40) and it resets the pointers to the beginning of both files before the serial READ# statements are executed. Here's the final printout –

```
Student ID#    Grade
 1111           88
 2222           67
 3333           98
 4444           81
 5555           99
```

Data printed in the file named "class" (see the program on the previous page) can be read by using the next program.

```
10 DIM A$[15]
20 FILES class
30 PRINT "  Name          Grade"
40 FOR I=1 TO 5
50 READ #1;A$,A
60 PRINT A$,A
70 NEXT I
80 END
```

Notice that the serial READ# statement must specify the types of data (data elements or string variable) in the order in which they were originally stored in the file. Line 50 reads a string variable and then a data point. This program can run only when the order of the data on file is known. Here's the printout –


```
  Name         Grade
Suzie Page       99
Blue Gill        90
Carol Rose       88
Jack Allison     74
Sean Thomas      80
```

The variables into which you read data items do not necessarily have to be the same variables from wich you printed the data items on the file. Although the variable name changes (from N$ and S, when stored, to A$ and A, when retrieved), the order in which the two data types are accessed is the same.

When a serial READ# statement encounters the EOF mark previously placed by the last PRINT# statement, the program ends and ERROR 93 indicates file overflow. The program can be written to end without displaying an error by using the IF END# statement described later in the chapter.

Repositioning the Record Pointer

As mentioned earlier, data pointers are automatically maintained. The pointers are automatically positioned at the beginning of the first record in a file after execution of a FILES statement or an ASSIGN statement. The word pointer is then automatically positioned at the next available storage location in the physical record after execution of a PRINT# statement. Finally, it is positioned at the next stored data item location of a physical record after execution of a READ# statement. The pointers are left unchanged in each file before execution of a serial PRINT# or READ# statement.

It's often necessary to position the record pointer to the beginning of a specific record in a file before executing a serial READ# statement. This is done by using only file number and record number parameters in a random READ# statement –

READ # file no. ; record no.

A serial PRINT# or READ# statement can then be executed after the record pointer has been repositioned, to access the beginning of the specified record, rather than the beginning of only the first record in the file.

To see how this works, first use the next program to store consecutive numbers beginning from the 8th record of a 15-record file named "data15" –

3

```
10 OPEN "data15",15
20 FILES data15
30 I=1
40 READ #1,8
50 PRINT #1,I
60 I=I+1
70 GOTO 50
80 END
```

The FILES statement sets the record pointer to the beginning of the first record in the file. The pointer is then repositioned to the beginning of the 8th record of data15 by line 40.

After printing into records 8 thru 15 of data15, ERROR 93 is displayed. This indicates that the physical end of file has been reached and no additional data can be printed in that file.

Now use this program to read the data, beginning at record 14 –

```
10 FILES data15
20 READ #1,14
30 READ #1;A,B,C,D,E,F,G,H
40 PRINT A;B;C;D;E;F;G;H
50 GOTO 30
60 END
```

The FILES statement automatically sets the record pointer to the beginning of the first record. The pointer is then repositioned to the beginning of record 14 by line 20. The serial READ# statement begins reading data from that point on.

Since each full-precision number uses 4 words of memory, 32 numbers can be printed into a 128-word record[1]. On the file data15, for example, the following numbers are stored on these corresponding records –

Record No.	Full-precision numbers
1 thru 7	(none)
8	1 thru 32
9	33 thru 64
10	65 thru 96
11	97 thru 128
12	129 thru 160
13	161 thru 192
14	193 thru 224
15	225 thru 256

3

The previous program read the data on records 14 and 15, and then output the data, eight numbers per row. Here's the printout –

193	194	195	196	197	198	199	200	record 14
201	202	203	204	205	206	207	208	
209	210	211	212	213	214	215	216	
217	218	219	220	221	222	223	224	
225	226	227	228	229	230	231	232	record 15
233	234	235	236	237	238	239	240	
241	242	243	244	245	246	247	248	
249	250	251	252	253	254	255	256	

ERROR 93 is displayed at this point, indicating that the physical end of file has been reached and there is no more data to be read. This error message can be avoided by using the IF END# statement, described later in the chapter.

[1] See Appendix B for details on how to estimate file size.

Random File Access

Data stored in a random manner is stored into specific physical records within a file. Variations of the previously discussed PRINT# and READ# statements are used to access data in particular records. As in serial file access, data pointers keep track of the data item currently being accessed. Unlike serial file access, however, in random file access, a specific record number within a file must be specified in each random PRINT# and random READ# statement. The record pointer is positioned at the beginning of the specified record before printing or reading occurs. Data is then printed or read consecutively from the beginning of the record.

The Random PRINT# Statement

PRINT # file no. , record no.[; data list [, END]]

PRINT # file no. , record no. ; END

The random PRINT# statement is used when data items are to be stored and accessed individually in specified records within a file.

The record number represents the location of a record in a specific file. This number can be an integer expression which does not exceed the number of records in the file. The data items can be variables, expressions, strings or substrings, or characters, and are printed from the beginning of the specified record.

The optional END parameter places an EOF (end of file) mark after the last data item printed in the last record. When END is not used, an EOR (end of record) mark is placed after the last item printed. In either case, the rest of the record is filled with EORs. The IF END# statement and the TYP function can be used to detect EOFs, as described later in the chapter.

The program below prints consecutive numbers onto each odd-numbered physical record of a 10-record file named TEN.

```
10 OPEN "TEN",10
20 R=A=1
30 FILES TEN
40 IF R>10 THEN 90
50 PRINT #1,R;A
60 A=A+1
70 R=R+2
80 GOTO 40
90 END
```

In line 50, the record number parameter is specified by the variable R. Line 70 increments this variable by 2 so that only odd-numbered records are accessed.

By printing in specific records of the file TEN, previous data in those records is erased and replaced by the new data. File TEN now contains –

Record No.	Data
1	1 (EOR)
2	(EOR)
3	2 (EOR)
4	(EOR)
5	3 (EOR)
6	(EOR)
7	4 (EOR)
8	(EOR)
9	5 (EOR)
10	(EOR)

The end of each odd-numbered record is automatically filled with EORs.

When neither the data list nor END are used in a PRINT# statement, it erases the contents of the specified record and fills it with EORs. For example, this program erases every third record of file TEN, which was opened and accessed in the previous program –

```
10 FILES TEN
20 FOR F=1 TO 10 STEP 3
30 PRINT #1,F
40 NEXT F
50 END
```

The information now left in the file is –

Record No.	Data
1	(EORs)
2	(EOR)
3	2 (EOR)
4	(EORs)
5	3 (EOR)
6	(EOR)
7	(EORs)
8	(EOR)
9	5 (EOR)
10	(EORs)

3

When an EOR is detected by either a random or serial READ# statement, it skips over the entire record and attempts to access data in the next record. You can use a PRINT# statement to write over the EOR marks.

When the data list is omitted from a PRINT# statement –

PRINT #file no. , record no. ; END

an EOF is placed at the beginning of the specified file; then the rest of the record is erased and filled with EORs. If a serial or random READ# then attempts to read from that file, reading the EOF causes ERROR 93.

The Random READ# Statement

READ #file no. , record no.[; data list]

The random READ# statement reads numbers and strings into variables from a specified record in a file, starting from the beginning of that record. String data and substrings can be read. A variation of this syntax can be used to reposition the record pointer, as shown later.

As with serial READ# statements, the variables into which you read data items do not have to be the same variables from which you printed the data items on the record, but they must be the same type and in the same order.

The following program reads the data printed in the 5th and 9th records of the file names TEN –

```
10 FILES TEN
20 READ #1,5;X
30 READ #1,9;Y
40 PRINT "Data in record 5 ="X
50 PRINT "Data in record 9 ="Y
60 END
```

3

This data was originally printed into odd-numbered records of file TEN (see page 37). The data in records 1 and 7 was erased. The program above reads the data from records 5 and 9 and outputs the data on the standard printer. If the program had specified to read every record in the file, however, ERROR 93 would appear after the EOR was read in record number 1. Here's the printout –

```
Data in record 5 = 3
Data in record 9 = 5
```

Logical File Access

Logical file access allows you to begin printing or reading data at any given word within a specified record of a file. This enables you to define subrecords or **logical records** within each physical record.

The Logical PRINT# Statement

PRINT # file no. ; record no. ; word pointer[; data list[; END]]

The logical PRINT# statement stores data items in a specified record of a file, beginning at the specified word. The file number and record number are the same as with a random file access. The word pointer can be an integer expression from 1 thru 129, and specifies where the first word of data is to be printed. 129 indicates the first word of the next record; so PRINT #1;1;129 is the same as PRINT #1;2;1.

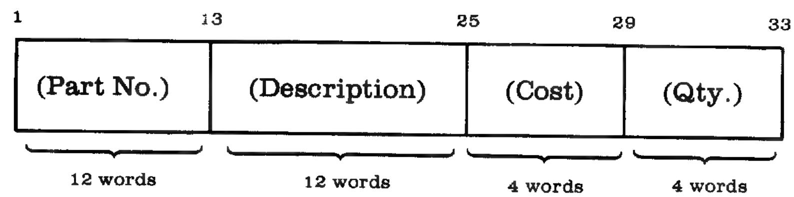
The optional END parameter places an EOF after the last data item printed. When END is not used, the remainder of the record is left unchanged. Remember that IF END# and TYP can be used to detect EOF's, as explained later in this chapter. When the data list and END parameters are not used, EORs are printed from the specified word to the end of that record.

Here's an example program which opens a 1,000-record file named "stock". Each record can contain 128 words of data about each part to be stocked; for now the program enters only four items: part number, description, unit cost, and current quantity on hand –

```
10 REM (OPEN STOCK FILE)
20 OPEN "stock",1000
30 DIM P$[20],D$[20]
40 FILES stock
50 PRINT LIN5,SPA30"PARTS SET UP"
60 PRINT "Part No.      Description"TAB40"Unit Cost     Qty. On Hand"LIN2
70 FOR I=1 TO 1000
80 DISP "PART NO.";
90 INPUT P$[1,20]
100 IF P$[1,4]="done" THEN 230
110 DISP "DESCRIPTION";
120 INPUT D$[1,20]
130 DISP "UNIT COST";
140 INPUT C
150 DISP "INITIAL QUANTITY";
160 INPUT Q
170 PRINT #1,I,1;P$
180 PRINT #1,I,13;D$
190 PRINT #1,I,25;C
200 PRINT #1,I,29;Q
210 PRINT P$,D$;TAB35;C,Q
220 NEXT I
230 PRINT "DONE"LIN5
240 END
```

Lines 50 thru 160 print headings for a table of input data, and then input the four items to be printed in each file; line 100 exits the input loop when the operator enters "done" for a part number. Lines 170 thru 200 print each item into logical records within the currently specified physical record (record I). Line 210 outputs the input data on the standard printer.

Notice that the word pointer parameter within each PRINT# (lines 170 thru 200) specifies the first word for each logical record. The data in each physical record looks like this –



These four items use only 32 words of each physical record; there are still 96 words available in each record for more data!

3

It's important to know the exact length of each string variable printed using logical PRINT#. For example, lines 90 and 120 in the last program generate 20-character strings, regardless of the number of characters input. This, in turn, ensures that 20-character (12 word) logical records will be printed in lines 170 and 180. If the subscripts were not used in lines 90 and 120, the first two logical records printed in each physical record would vary in size, depending on the current string length. This would change the location (first word) of each successive logical record!

The Logical READ# Statement

READ # file no. , record no. , word pointer[; data list]

The logical READ# statement reads numbers and strings into variables from a specified record, starting from a specified word. Strings and substrings can be read.

As with serial and random READ# statements, the variables into which you read data items do not necessarily have to be the same variables from which you printed the data items on the record, but they must be the same type and in the same order as the originals. When the data list is not used, the logical READ# resets the record pointer and word pointer to the specified record and word.

The previous example program opened a file to hold information on parts to be stored. The next program opens a file named "ordrpt" (for "order point") and then searches the file "stock" for any item with a quantity-on-hand of less than 10 (lines 70 thru 100). When such an item is found, lines 120 thru 140 read the part number, print the part number and quantity in record I of file "ordrpt", and output the same data on the standard printer –

```
10 REM (ORDER POINT REPORT)
20 OPEN "ordrpt",200
30 DIM P$[20],D$[20]
40 FILES stock,ordrpt
50 PRINT LIN5,SPA20,"PARTS TO REORDER"
60 PRINT "Part No.        Qty. On Hand"LIN2
70 FOR I=1 TO 1000
80 READ #1,I,29;Q
90 IF Q<10 THEN 120
100 NEXT I
110 GOTO 160
120 READ #1,I,1;P$,D$
130 PRINT #2;P$,Q
140 PRINT P$;D$;TAB35;Q
150 NEXT I
160 PRINT "DONE"LIN5
170 END
```

3

Notice that logical READ# (lines 80 and 120) enables you to read only the items required from a record, thus saving memory space and program execution time. Data is printed into file "ordrpt" serially, since it need not be accessed separately.

As another example of logical file access, the next program can be used to update the cost and quantity data for each record of file "stock" used in the previous programs. Lines 40 thru 130 print headings for a table and input data to update each record. As in the first program, this program exits the input loop when the operator inputs "done" for a part number.

After the operator enters each set of data, a subroutine (lines 210 thru 290) searches the file for the appropriate record. After the record is found, lines 150 and 160 print the new list and quantity into that record. Line 170 then outputs the new data on the standard printer.

```
10 REM (UPDATE STOCK FILE)
20 DIM P$[20],N$[20]
30 FILES stock
40 PRINT LIN5,SPA20"PARTS RECEIVED"
50 PRINT "Part No.         Cost       Qty. Received   Qty. On Hand"LIN2
60 FOR I=1 TO 1000
70 DISP "PART NO.";
80 INPUT P$[1,20]
90 IF P$[1,4]="done" THEN 190
100 DISP "QUANTITY RECEIVED";
110 INPUT Q1
120 DISP "UNIT COST";
130 INPUT C1
140 GOSUB 210
150 PRINT #1,I,25;C1
160 PRINT #1,I,29;Q+Q1
170 PRINT P$,C1,Q1,Q+Q1
180 NEXT I
190 PRINT "DONE"LIN5
200 END
210 REM (FILE SEARCH ROUTINE)
220 FOR J=1 TO 1000
230 READ #1,J,1;N$
240 IF P$=N$ THEN 290
250 NEXT J
260 DISP "PART NOT ON FILE!";
270 WAIT 2000
280 GOTO 70
290 RETURN
300 END
```

3

The IF END# Statement

IF END # file no. THEN line no.

The IF END# statement sets up a branching condition in the program. If the physical end of file is encountered during a PRINT# or READ# statement, or if an EONV MARK IS ENCOUNTERED DURING A READ# statement, the program branches to the line number specified. This avoids ERROR 93 and makes it possible to use a file whose exact contents are unknown.

The IF END# statement will also cause a branch out of a random or logical PRINT# or a READ# when the physical end of record is seen.

3

The IF END# statement may be executed from the keyboard, but READ# and PRINT# operations executed from the keyboard will not reference it.

In some of the previous programs, for example, ERROR 93 appears after the last itm is accessed, telling you that the physical end of file has been reached. This error message can be avoided by including an IF END# statement in the program. Here is a modified version of the program shown on page 35. The program branches to line 70 when the end of file is reached.

```
10 FILES data15
20 READ #1,8
30 IF END#1 THEN 70
40 READ #1;A,B,C,D,E,F,G,H
50 PRINT A;B;C;D;E;F;G;H
60 GOTO 40
70 PRINT LIN2"END OF DATA REACHED!"LIN2
80 END
```

Notice that IF END# is executed **before** entering the serial READ#/PRINT# loop. Since IF END# established the exit procedure for this loop, it has to be executed before entering the loop, but should not be included in the loop. Repeated execution of IF END# should be avoided, since it will only increase program execution time.

NOTE

An IF END# statement sets up a condition to detect an EOF mark. If you attempt to access a non-existent or invalid record without a previously executed IF END#, ERROR 93 is displayed. If a file has not been assigned into an # position which is referenced, or no FILES statement is given, executing the IF END# results in ERROR 90.

3

If the line number to which the IF END# statement refers does not exist, ERROR 44 is displayed. This error refers to a PRINT# or READ# statement, not the IF END# statement, since the PRINT# or READ# **caused** the error.

As another example, this program prints four data items into each record of the file "pwr10". When variable A is incremented to a value greater than 10 (the number of records in the file) the condition set up by the IF END# statement is met. In this case, the program branches to line 80 when the physical end of file is encountered. In this way, ERROR 93 is avoided.

```
10 OPEN "pwr10",10
20 FILES pwr10
30 A=1
40 IF END#1 THEN 80
50 PRINT #1,A;A,A^2,A^3,A^4
60 A=A+1
70 GOTO 50
80 PRINT LIN2"FILE pwr10 FILLED!"LIN2
90 END
```

The TYP Function

TYP file no.

TYP (– file no.)

The TYP (type) function identifies the type of the next data item to be accessed in a specified file. It returns one of these codes –

Type Code	Meaning
0	Item not printed via the 9831A.
1	Next item is a full-precision number.
2	Next item is a string variable contained in one record.
2.1	First part of a multi-record string.
2.2	Intermediate part of a multi-record string.
2.3	Last part of a multi-record string.
3	Next item is an EOF or the physical end of file.
4	Next item is an EOR or the physical end of record.
5	Next item is a split-precision number.
6	Next item is an integer-precision number.

When the file number is a positive number, TYP advances the word and record pointers to the data item type returned. To not advance the word pointer, use a negative file number.

When a file number of 0 is given, the data type of the next DATA item to be read via the next READ statement in the program is returned.

The program on the next page shows how to use TYP. But first run this program to print (serially) various types of data on a new file named "type?" –

```
10 OPEN "type?",5
20 FILES type?
30 DIM A,B$[10],CS,DI
40 A=1111
50 B$="STRING"
60 C[1]=2222
70 D[1]=3333
80 PRINT #1;A[1],B$,C[1],D[1]
90 END
```

Now run this program to identify each data item and then store it into the appropriate type of variable –

```
10 FILES type?
20 DIM A,B$[10],CS,DI
30 READ #1,1
40 GOTO TYP(-1) OF 50,80,110,130,150,180
50 READ #1;A
60 PRINT A"is a full-precision number."
70 GOTO 40
80 READ #1;B$
90 PRINT B$" is a sting variable."
100 GOTO 40
110 PRINT "EOF mark is next."
120 GOTO 210
130 PRINT "EOR mark is next."
140 GOTO 210
150 READ #1;C[1]
160 PRINT C[1]" is a split-precision number."
170 GOTO 40
180 READ #1;D[1]
190 PRINT D[1]" is an integer-precision number."
200 GOTO 40
210 END
```

Line 30 sets the record pointer to record 1 of file "type?". The computed GOTO statement (line 40) branches the program to one of six line numbers, depending upon the value returned by `TYP(-1)`. This statement is executed before the READ# to determine which type of data is to be read next. Here's the printout –

```
 1111     is a full-precision number.
STRING is a string variable.
 2222     is a split-precision number.
 3333     is an integer-precision number.
```

Notice that if the record pointer had been set to any other record of file "type?" (for example, READ #1,4) the TYPE function would return 3, indicating an EOF mark. Remember that each record is filled with EOF's when it is opened; the EOFs are replaced by data via PRINT# statements.

As shown in the last program, TYP is the means of checking for EOF and EOR marks. While an IF END# statement sets up a branching condition for either mark during a random or logical access operation, IF END# can only check for EOF marks during a serial access operation. TYP(-1) is the only method available to check for EORs. Be sure to use a negative file number when checking for EOR marks, to avoid advancing the word pointer.

3

The SLEN Function

SLEN file no.

The SLEN (string length) function is used to check for strings in the specified file. If the next item is a string variable, its length, in characters, is returned. −1 is returned if the next item is not a string.

The SIZE Function

SIZE file no.

The SIZE function returns the size, in records, of a specified file. This is the same as the LENGTH value printed via the CAT (catalog) statement.

The REC Function

REC file no.

The REC (record) function returns the current position of the record pointer within the specified file. The record pointer is described on page 27.

The WRD Function

WRD file no.

The WRD (word) function returns the current position of the word pointer for the specified file. The value can be from 1 thru 129; 1 indicates the first word of the record, while 129 indicates that the next item to be printed will go into the first word of the next record.

Chapter 4
Matrix Operations

The matrix operations available with the Flexible Disk ROM enables you to initialize data matrices, and then store and read matrices using disk storage. These matrix statements are available –

MAT PRINT#	Print data matrices from array variables into a specified file.
MAT READ#	Read data matrices from the disk into array variables.
MAT ZERO	Set all specified array elements to 0.
MAT CON	Set all array elements to a specified value.
REDIM	Change the current working size of an array for matrix operations.

Introduction to Matrices

A table of data, or any collection of data elements arranged in rows and columns, is known as a **matrix**. A list of data, or any collection of data elements arranged in a single row or column, is known as a **vector**.

Here is an example of a matrix –

Grade	Boys	Girls
1	10	7
2	9	8
3	9	10
4	7	9
5	7	10
6	9	11

Here is an example of a vector –

```
Test Scores
 93
 85
 79
 89
 69
 95
 100
```

Dimensioning Matrices

Matrices are stored in array variables. Since arrays can be named from A thru Z, up to 26 matrices can be assigned in one program at a time.

To reserve storage space for arrays, a DIM or COM[1] statement is used. Arrays not listed in a DIM statement are assumed to have 10 elements if they are one-dimensional, or 10 rows and 10 columns if they are two-dimensional.

The number of elements in a one-dimensional array (vector) or the number of rows or columns in a two-dimensional array (matrix) can be specified as an integer from 1 thru 256. For example –

```
10 DIM A[100]
20 DIM B[75,30]
30 DIM C[30,30]
```

Line 10 dimensions array A, a column vector of 100 row elements. Line 20 dimensions array B to 75 rows by 10 columns. This row-column convention exists throughout the manual. Line 30 dimensions array C, which has the same number of rows and columns.

Filling Matrices

The INPUT and READ (with DATA) statements are used to enter values into matrices. For example, this sequence inputs a 5 by 3 matrix, one row at a time –

```
10 DIM A[5,3]
20 FOR J=1 TO 5
30 INPUT A[J,1],A[J,2],A[J,3]
40 NEXT J
50 END
```

[1] The COM statement must be the first statement stored in a program. Refer to the 9831A Operating and Programming Manual for more details on COM.

As another example, you can include the values to be stored in a matrix into a DATA statement, and then use READ, with a for-next loop, to fill the matrix –

```
10 DIM G[6,2]
20 DATA 10,7,9,8,9,10,7,9,7,10,9,11
30 FOR C=1 TO 6
40 FOR R=1 TO 2
50 READ G[C,R]
60 NEXT R
70 NEXT C
```

The PRINT statement allows you to print headings and other information with a matrix. This sequence, for example, prints the 6 by 2 matrix (A) filled above –

```
80 PRINT "Grade           Boys            Girls"
90 FOR R=1 TO 6
100 PRINT R,G[R,1],G[R,2]
110 NEXT R
120 END
```

The printout is on page 51.

The HP 98223 Matrix/Plotter ROM provides MAT READ and MAT PRINT statements for filling matrices from DATA lists and printing matrices on the standard printer. The ROM also has many other matrix operations, such as matrix arithmetic, inversion and transposition. Refer to the Matrix/Plotter ROM Programming Manual for details.

The MAT PRINT# Statement

MAT PRINT # file no. [; record no.[; word pointer]] ; matrix list

The MAT PRINT# statement prints an entire matrix into a specified record or file. MAT PRINT# routines are easier to program, require less memory and execute faster than individual PRINT# statements.

Matrices can only be printed into data files. The size of the matrix you want to store is limited by the number of records you specify when opening that file. For example, a one-record file, which contains 128 words, can hold up to 32 full-precision numbers. This means that a 4 by 8 matrix is the largest matrix of full-precision numbers that one record can hold. Of course, by printing a matrix into a multi-record file, the matrix size is not limited to 128 words. In this case it is limited by the number of records in the file multiplied by 128 words per record.

This program, for example, prints a 4 by 4 matrix serially into a file named "matrix" –

```
10 OPEN "matrix",5
20 DIM A[4,4]
30 FILES matrix
40 FOR I=1 TO 4
50 FOR J=1 TO 4
60 A[I,J]=10*I+J
70 NEXT J
80 NEXT I
90 MAT PRINT # 1;A
100 END
```

The elements of a matrix are printed consecutively, in row-column order, from the beginning of the file or record specified. When the record number is omitted, the matrix is printed into the file from the position of the record pointer. By including this optional parameter, however, the matrix is printed beginning at the specified record. Any number of records can be printed.

4

If the matrix is too large for the file specified, an EOF mark will be encountered and ERROR 93 will be displayed. Of course, an IF END# statement can be used to detect the EOF and avoid the error.

The MAT READ# Statement

MAT READ # file no. [, record no.[, word pointer]] ; matrix list

The MAT READ# statement reads the matrix from a specified record or file. MAT READ# uses less memory and executes faster than individual READ# statements. For example, to read the matrix created in the previous example program and print it on the standard printer, use this program –

```
10 DIM B[4,4]
20 FILES matrix
30 MAT READ # 1;B
40 FOR R=1 TO 4
50 PRINT B[R,1],B[R,2],B[R,3],B[R,4]
60 NEXT R
70 END
```

Line 30 reads the matrix from file "matrix" into the memory. Lines 40 thru 60 then prints the entire matrix. Notice that although array A was used to define the original matrix, any array can be used to read it back again (in this case, B is used). This array must be dimensioned as in line 10. The printout is on the next page.

```
11          12          13          14
21          22          23          24
31          32          33          34
41          42          43          44
```

A matrix can be read in any format less than or equal to the original size. In the next program, for example, a 4 by 4 matrix is read as an 8 by 2 matrix by dimensioning the array variable in that manner.

```
10 DIM B[8,2]
20 FILES matrix
30 MAT READ # 1;B
40 FOR R=1 TO 8
50 PRINT B[R,1],B[R,2]
60 NEXT R
70 END
```

Here's the printout –

```
11          12
13          14
21          22
23          24
31          32
33          34
41          42
43          44
```

Notice that data element (2,1) for example, is 13. This is because the data elements of the original matrix were printed on the file point by point; they were not stored in unique randon locations.

If you run a program in which a matrix was dimensioned larger than the original matrix, ERROR 93 is displayed. This error message can be avoided, however, by using an IF END# statement to detect the end of record.

The MAT READ# statement will automatically redimension the specified array when subscripts are used. For example, to read the data from file "matrix" into a 3 by 5 format, use this program –

```
10 DIM C[4,4]
20 FILES matrix
30 MAT READ # 1;C[3,5]
40 FOR R=1 TO 4
50 PRINT C[R,1];C[R,2];C[R,3];C[R,4];C[R,5]
60 NEXT R
70 END
```

Notice that the DIM statement (line 10) dimensions an array that is at least as large as that specified by the MAT READ#. The printout is on the next page.

```
11    12    13    14    21
22    23    24    31    32
33    34    41    42    43
```

The MAT ZERO Statement

MAT ZERO array list

The MAT ZERO statement sets all elements in the specified array(s) to 0. If array subscripts are used, the array is automatically redimensioned to the specified working size before being zeroed.

For example, this sequence –

```
200 DIM Z[5,5]
210 MAT ZERO Z[5,3]
```

initializes this array and matrix –

$$\begin{array}{cc|ccccc} & & \multicolumn{5}{c}{\text{(column)}} \\ & & 1 & 2 & 3 & 4 & 5 \\ \hline & 1 & 0 & 0 & 0 & & \\ & 2 & 0 & 0 & 0 & & \\ \text{(row)} & 3 & 0 & 0 & 0 & & \\ & 4 & 0 & 0 & 0 & & \\ & 5 & 0 & 0 & 0 & & \end{array}$$

Use of the subscripts in MAT ZERO specifies a matrix working size of 5 by 3; in this case, columns 4 and 5 are left undefined; matrix operations would not access these undefined columns unless specified by subscripts.

Since 0 is the logical value for "false", a zeroed matrix is useful for logic initalization. For example, assume that the first record of a file called "grades" has these 20 scores for students in a programming course –

Student	Grade
1	93
2	85
3	79
4	89
5	68
6	95
7	100
8	66
9	79
10	85

Student	Grade
11	80
12	66
13	52
14	79
15	98
16	95
17	89
18	68
19	72
20	96

This program can be used to read the matrix from the file into array G and then count the number of students receiving each grade –

```
10 DIM A[100],G[20]
20 FILES grades
30 MAT READ # 1;G
40 FOR I=1 TO 20
50 G=G[I]
60 A[G]=A[G]+1
70 NEXT I
80 PRINT "Grade     Number of Students"
90 FOR I=1 TO 100
100 IF A[I]=0 THEN 120
110 PRINT I,A[I]
120 NEXT I
130 END
```

After array G is filled by the matrix stored in file "grades", line 40 zeros a second array. The first for-next loop then records each element of array G and increments an element of array A corresponding to the grade's value; this array is the grade counter. The second for-next loop scans array A and prints only the non-zero elements as the number of students receiving each grade. Here's the printout –


```
Grade     Number of Students
 52            1
 66            2
 68            2
 72            1
 79            3
 80            1
 85            2
 89            2
 93            1
 95            2
 96            1
 98            1
 100           1
```

The MAT CON Statement

MAT CON expression ; array list

The MAT CON (constant) statement initializes each element of each specified array to the value of the expression. When array subscripts are used, the array is automatically redimensioned to the new working size before being initialized. For example, this statement sets all elements of array A to 1 –

```
MAT CON 1;A
```

The REDIM Statement

REDIM array list

4

The REDIM (redimension) statement changes the matrix working size of the specified array(s) to new specified boundaries. The subscripts specifying the new working size must be equal to or larger than the original array size. (The working size of a matrix is the same as the physical array size unless it is redimensioned.) When subscripts are not used, the working size is redimensioned to the array's original size.

When a new working size is specified for an array containing data, the data is not erased, but rearranged. When the new working size is smaller than the original size, any data not included in the new size is not lost, but it is currently inaccessible via matrix operations (see the next examples). This data is available again when the array is redimensioned to its original size.

The array's matrix working size can also be redimensioned by using subscripts in MAT READ#, MAT ZERO, and MAT COM statements. For example, this program first dimensions a 10 by 10 array and then initializes a 5 by 6 matrix to zero –

```
10 DIM A[10,10]
20 MAT ZERO A[5,6]
30 FOR R=1 TO 5
40 FOR C=1 TO 6
50 A[R,C]=R*C
60 NEXT C
70 NEXT R
80 FOR I=1 TO 5
90 PRINT A[I,1];A[I,2];A[I,3];A[I,4];A[I,5];A[I,6]
100 NEXT I
```

Lines 30 thru 100 fill the matrix with integer data and then print the matrix –

```
1     2     3     4     5     6
2     4     6     8     10    12
3     6     9     12    15    18
4     8     12    16    20    24
5     10    15    20    25    30
```

The next sequence redimensions the matrix and prints it again –

```
110 REDIM A[10,3]
120 FOR I=1 TO 10
130 PRINT A[I,1];A[I,2];A[I,3]
140 NEXT I
```

Matrix A now has a working size of 10 by 3. Notice that the data is not lost, but rearranged to fill the new working size –

```
1     2     3
4     5     6
2     4     6
8     10    12
3     6     9
12    15    18
4     8     12
16    20    24
5     10    15
20    25    30
```

This sequence stores the 10 by 3 matrix into a previously-opened file named "redim" –

```
150 FILES redim
160 MAT PRINT # 1;A
```

The last sequence reads the matrix back into array A and prints the data –

```
170 MAT READ # 1,1;A[3,10]
180 FOR R=1 TO 3
190 FOR C=1 TO 10
200 PRINT A[R,C];
210 NEXT C
220 PRINT
230 NEXT R
240 END
```

Notice that the matrix working size is changed to 3 by 10 in line 170, so the matrix now looks like this –

1	2	3	4	5	6	2	4	6	8
10	12	3	6	9	12	15	18	4	8
12	16	20	24	5	10	15	20	25	30

Chapter 5
Additional Operations

Introduction

The statements described in this chapter are –

UNIT	Specify the disk drive to be used for subsequent operations.
PRINT LABEL, READ LABEL	Identify each disk by name.
DCOPY	Duplicate existing disk files.
DREN	Rename an existing disk file.
DSAVE, DGET	Store and load source programs.
DBYTE	Convert an expression to a string character.
DEXP	Convert an expression to a four-character string.
CERROR	Cancel the SERROR (error recovery) condition.
UCASE	Convert a string to uppercase.
WCTL	Output a control number to an interface card.

These functions are described here –

FRAC	Return the fractional part of a number.
LEX	Compare two strings by their value.
NUM	Return the numerical value of a string character.

(continued)

UPOS Return the position of one string within another, regardless of whether the letters are in uppercase or lowercase.

STD Return the current select code specified by STDPR (standard printer).

The UNIT Statement

UNIT drive no. [; select code]

The UNIT statement specifies the disk drive number and optionally, the drive select code, to be used by subsequent disk operations. This statement changes the drive number (0) and the select code (8) automatically specified for disk operations when the desktop computer is switched on. Drive numbers 0 thru 3 and select codes 8 thru 15 can be used.

As mentioned eariler, up to three 9885S (slave) disk drives can be connected via each 9885M (master) drive. If needed, up to eight[1] 9885M drives can be connected to the desktop computer. Each set of drives consists of a master and its slave drives, and all respond to the select code set on the master drive's interface card (set to select code 8 at the factory). So each drive in the set must respond to a different drive number. The drive number switch is on the back of each drive. Instructions for setting the switch are in Appendix A.

Once a UNIT statement has been executed, all subsequent disk operation which do not specify a drive number are addressed to the new drive number and select code. Subsequent UNIT statements can be used to change the drive number and select code again. The UNIT drive number and select code are canceled when ERASE A or LOAD BIN is executed or when the desktop computer is switched off (drive number 0 and select code 8 are reset).

Executing a UNIT statement with a select code other than that previously set automatically erases the current FILES list of file names and numbers (see The FILES Statement in Chapter 3).

[1] An HP 9878A I/O Expander enables more than three peripheral devices to be connected to the desktop computer.

For example, to transfer a program stored in a file called "print A", which is on the disk in drive no. 0, to a new file called "print B" on the disk in drive no. 1, use these statements –

```
GET "print A"

UNIT 1

SAVE "print B"
```

Now assume that you have three data items stored on the fifth record of the file named "give" on drive 1. To copy this data from that file to the sixth record of a new file named "take" on drive 2, run this program –

```
10 UNIT 2
20 OPEN "take",10
30 FILES give:1,take:2
40 READ #1,5;A,B,C
50 PRINT #2,6;A,B,C
60 END
```

5

Notice that specifying drive numbers in the FILES list allows you to easily access more than one drive.

Disk Labels

The next two statements allow you to assign a name to each disk and then identify the disk by name.

The PRINT LABEL Statement

`PRINT LABEL;` drive no. `;` "text" or string variable

This statement assigns an alphanumeric label to the disk in the specified drive. The label is stored in the Systems Table on the disk (see Disk Structure in Chapter 1). The label can be up to 224 characters long.

For example, to assign the label "database" to the disk in drive no. 0, execute –

```
PRINT LABEL;0;"database"
```

The READ LABEL Statement

READ LABEL, drive no., string variable

This statement reads the label on the disk and stores it in the specified string or substring. If the disk doesn't have an assigned label, the null string is returned.

Here's a program sequence which reads and prints the current disk's label –

```
90   :
100 DIM A$[224]
110 READ LABEL,0,A$
120 IF A$="" THEN 150
130 PRINT A$" is in drive no. 0."LIN2
140 GOTO 160
150 PRINT "The disk in drive no. 0 is unlabeled."LIN2
160  :
```

The DCOPY Statement

5

DCOPY file name [, drive no.] TO file name [, drive no.]

The DCOPY statement duplicates the contents of one file into another. The optional drive numbers allow you to copy a file from one drive to another.

Non-Data Files

When copying non-data files, the second file is automatically initialized to the same size as the first file; if the second file name already exists, however, the operation is canceled with ERROR 92. As an example, this statement could be used in place of the first example sequence on page to transfer a program from a file on drive 0 to a new file on drive 1 –

```
DCOPY "print A",0 TO "print B",1
```

An advantage of using DCOPY here is that the UNIT drive number and select code have not been changed.

If a checkword error occurs while a non-data file is being copied, the operation is canceled.

Data Files

When copying data into a new file, the second file name is automatically opened with the same size as the first file. After the new file is opened, the data is copied and any remaining space is filled with EOF marks. This statement, for example, opens a new data file called "take" on drive 2 and copies the data from an already existing file named "give" on drive 1.

```
DCOPY "give",1 TO "take",2
```

Before copying data into a file name which already exists, the size of the second file is checked. If it is larger than the first file, the data is copied and the rest of the file is filled with EOFs. If the second file is smaller than the first, however, data will be copied only if all extra records in the second file are filled with EOFs or EORs. Otherwise, the operation is canceled. When data is copied into a smaller file than the original, data is copied until the smaller file is filled.

If a checkword error occurs while a data file is being copied, the operation will be completed before the error message is displayed.

The DREN Statement

DREN old file name TO new file name

5

The DREN (rename) statement allows you to change the name of any file. The contents of the file remain the same.

For example, here's a program which changes file names on the currently set drive number –

```
10 DIM A$[6],B$[6]
20 DISP "PRESENT FILE NAME";
30 INPUT A$
40 ASSIGN A$,1,Z
50 IF Z=3 THEN 120
60 DISP "NEW FILE NAME";
70 INPUT B$
80 ASSIGN B$,1,Z
90 IF Z#3 THEN 140
100 DREN A$ TO B$
110 END
120 PRINT A$" does not exist...is it spelled correctly?"LIN2
130 GOTO 20
140 PRINT B$" is already in use...select another name."LIN2
150 GOTO 60
160 END
```

As shown in line 100, string variables can be used for file names in DREN.

The DSAVE Statement

DSAVE file name [, 1st line no.[, 2nd line no.]]

The DSAVE statement stores a program as a **source** program, a series of data strings, in the specified file. The file must have previously been opened. The optional line numbers allow you to store selected lines of the program currently in the memory.

The DGET statement is used to reload a source program back into the memory. The SAVE and GET statements (see Chapter 2) cannot be used to handle source programs.

A source program is printed sequentially into a data file as a series of string variables, one BASIC statement per string. After the last statement (string), an EOF mark is written to separate old data from the source program.

The DGET Statement

DGET file name [, 0]

5

The DGET statement loads a source program previously stored with DSAVE back into the memory. The program is syntax-checked as it is loaded. The machine begins running the program after it has been loaded. The optional 0 parameter is used to prevent the program from being automatically run.

The DBYTE Statement

DBYTE expression , string variable

The DBYTE statement converts the value of the expression to its binary equivalent character. This binary character is then stored as a single character in the specified string or substring. The value stored is the decimal equivalent of an ASCII character. For example, if you set Y equal to 32, A$ will contain the quotation mark (decimal 34) by executing –

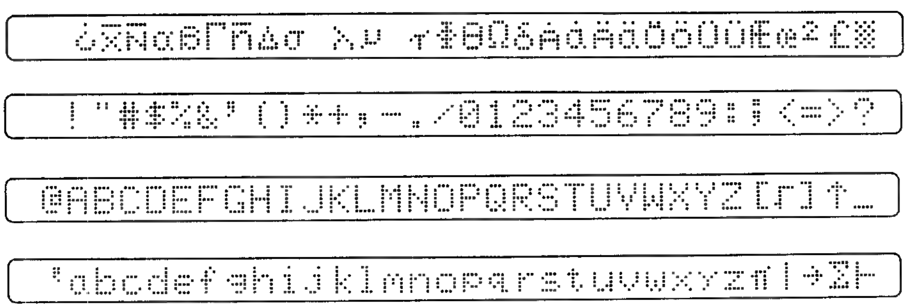
```
DBYTE Y+2,A$
```

A table of ASCII-decimal values is in Appendix B.

As another example, here's a short program which generates a string containing each of the characters in the display character set –

```
10 DIM A$[129]
20 FOR I=0 TO 128
30 DBYTE I,A$[I+1]
40 NEXT I
50 DBYTE 32,A$[1,1]
60 DBYTE 32,A$[11,11]
70 DBYTE 32,A$[14,14]
80 FOR I=1 TO 128 STEP 32
90 DISP A$[I]
100 WAIT 5000
110 NEXT I
120 END
```

Since ASCII-decimal values 0 (null), 13 (carriage return), and 10 (linefeed) are not used with the display, those values are replaced by decimal 32 (character spaces) in lines 50 thru 70. The displays are –



The DEXP Statement

DEXP expression , string variable

The DEXP statement converts the value of the specified expression into a 4-digit character string with leading zeros. DEXP can be used, for example, to generate line numbers for BASIC statements. If you set X equal to 10, the first four characters of A$ will contain the string 0010 by executing –

```
DEXP X,A$[1,4]
```

The CERROR Statement

CERROR

The CERROR (clear error) statement cancels any error recovery routine set by SERROR. The SERROR statement is described in Chapter 3 of the 9831A Operating and Programming Manual.

The UCASE Statement

UCASE string variable

The UCASE (uppercase) statement convert all characters in the specified string or substring to uppercase.

For example, this program sequence asks the operator for a YES or NO answer, and then inputs the reply. Line 100 ensures that the reply is in uppercase –

```
     :
70   :
80 DISP "DO YOU WANT A REPORT (YES OR NO)";
90 INPUT A$
100 UCASE A$
110 IF A$="NO" THEN 250
120 }
    }  print report
240 }
250  :
     :
```


The WCTL Statement

WCTL select code ; expression

The WCTL (write control) statement outputs a binary number for controlling various functions on the specified interface card. The number sets a combination of bits on the interface card's control register, R5. The WTCL statement controls the lower four bits on the control register, bits 0 thru 3. Each bit has a unique binary weight, from 1 (the first or least-significant bit) to 8 (the 4th bit). Although the expression can range from 0 thru 32767, its effective range is only from 0 thru 15. An introduction to binary coding is in the 9831A Peripheral Control Manual.

The control bits available with each HP interface card are described in its Installation and Service Manual. For example, these control bits are available with the HP 98032A Interface (the interface used with the 9885M Disk Drive) –

7	6	5	4	3	2	1	0
Not Controlled				N/A	N/A	CTL 1	CTL 0

98032A Control Bits with WCTL

Control bits 0 and 1 are used to transfer information to a peripheral device. These bits are automatically used with the 9885 Disk Drive and must not be controlled with WCTL.

Control bits 2 and 3 are not used on the 98032A, but may be used on other interface cards; again, refer to the interface installation and service manual.

Additional Functions

The following functions enhance string and data-handling capability.

The FRAC Function

`FRAC` expression

This function returns a fractional part of the expression. Here are some examples –

`FRAC 1.3`	`.3`
`FRAC (-1.1)`	`.9`
`FRAC (1)`	`0`

Notice that the fraction, when added to the integer (INT) part of the expression, returns the original value. Using the second example –

$$X=INT(X) + FRAC(X) = -2 + .9 = -1.1$$

The LEX Function

LEX (string variable, "text" or string variable)

The LEX (lexicon) function compares the two strings, character-by-character, according to the ASCII value of their characters. If the first string is greater than the second, 1 is returned. If the strings are of equal value, 0 is returned. If the first string is less than the second, −1 is returned.

Here are some examples (assume that A$ has been dimensioned as a 10-character string) –

A$="ABC"	ABC	
LEX (A$,"ABD")	−1	(A$<"ABD")
LEX (A$,"AB")	1	(A$>"AB")
LEX (A$,"ABC")	0	(A$="ABC")
LEX (A$,"ABCD")	−1	(A$<"ABCD")

5

Here's a program line that branches the program to line 20 when the value of A$ is greater than B$ –

```
90 IF LEX (A$,B$)>0 THEN 200
```

The NUM Function

NUM (string variable)

The NUM (number) function returns the ASCII-decimal value of the first character of the specified string or substring. If the string is a null string, 0 is returned. For example, assuming that A$ is still dimensioned and assigned as in the previous example –

A$	ABC	
NUM (A$)	65	
NUM (A$[3])	67	
NUM (A$[4])	0	(null string)

The UPOS Function

UPOS (string variable , "text" or string variable)

The UPOS (uppercase position) function returns the position of the first character of the second string within the first string. The strings are temporarily converted to uppercase before being compared. The strings are returned to their original state after the position is returned.

UPOS allows you to find the first occurrence of a specified string within another string, regardless of whether the characters are in lowercase or uppercase. For example, assume that A$ is still dimensioned as a 10-character string –

A$="Strings"	Strings
UPOS (A$,"Ring")	3
UPOS (A$,"s")	1
UPOS (A$[2],"s")	7

The STD Function

STD

The STD (standard) function returns the standard printer select code currently specified by the STDPR statement. Select code 2 is automatically set when the desktop computer is switched on or when ERASE A is executed.

For example, if you wish to use the WRITE statement to control the standard printer, but the currently set STDPR select code is not known, this method could be used –

WRITE (STD, line no.)...

STD is used here in place of the select code parameter.

The WRITE and STDPR statements are explained in Chapter 8 of the 9831A Operating and Programming Manual.

Appendix A
Installation and Service

Unpacking Your System

You should have already carefully removed your desktop computer, 9885M Drive, and 9885S Drive(s) if ordered, from their shipping packages. After unpacking the drive(s), remove the foam shipping piece from the drive door.

Equipment Supplied

Check to be sure that the following equipment is supplied with your Disk Drive. Notice that some items are supplied only with each 9885M Drive.

Equipment Supplied

Description	9885M	9885S	Part Number
Disk Operating and Programming Manual	1	0	09885-90050
Disk Care Note	1	1	09885-90020
Flexible Disk ROM	1	0	HP 98218A
Initialized Disk	1	0	09885-90060
Blank Disk	1	2	[1]
9885S Interface Cable	0	1	09885-61607
HP 98032A Interface Card	1	0	HP 98032A Opt. 185
Power Cord (USA)[2]	1	1	8120-1378
Spare Fuses			
(3 amp for 110 volts)	1	1	2110-0381
(2 amp for 220 volts)	1	1	2110-0303
Fuse Cap, European	1	1	2110-0544
Drive Number Labels (0 thru 3)	1 Set	1 Set	7120-5160
Select Code Labels (8 thru 15)	1 Set	1 Set	7120-5161
Disk Labels	1 Set	1 Set	7120-5330
WRITE Tabs	1 Sheet	1 Sheet	7120-5388
Notebook	1	0	9282-0580

1 Blank disks may be ordered in packages of five using part number 09885-80004.

2 Other power cords are shown on page 76.

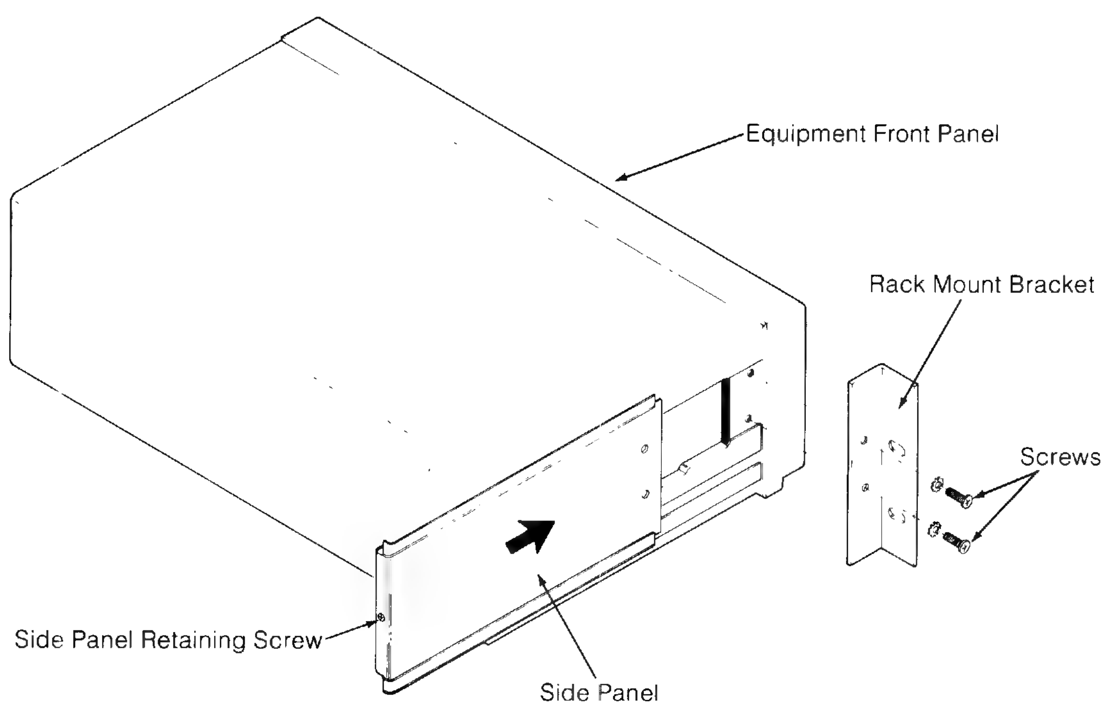
Option 002 Rack Mount Kit

This option allows you to mount your drive or an HP 9878A I/O Expander in a standard 19-inch rack mount cabinet. This option is installed at the factory, although a rack mount field installation kit is available.

Rack Mount Installation

The rack mount brackets are not able to support the entire weight of the equipment. A shelf or other support should be provided by the equipment rack or cabinet to support the weight.

To install the rack mount kit, first replace the standard side panels with those supplied in the rack mount kit (refer to the next figure). Then install the rack mount brackets with the screws provided in the kit.



Rack Mount Kit Installation

Checking Fuses, Voltage and Power Cords

Fuses

Always be sure that the correct fuse is installed. Failure to follow this precaution may result in damage to the drive.

A different fuse is required for each of the two voltage ranges of 100-120 Vac and 220-240 Vac. Be sure that the fuse on the rear panel is the proper type and rating, as shown below.

Fuses

Voltage Setting	Fuse Rating	HP Part Number
100, 120	3 amp (SB)	2110-0381
220, 240	2 amp (SB)	2110-0303

WARNING

TO AVOID THE POSSIBILITY OF SERIOUS INJURY, DISCONNECT THE AC POWER CORD BEFORE REMOVING OR INSTALLING A FUSE.

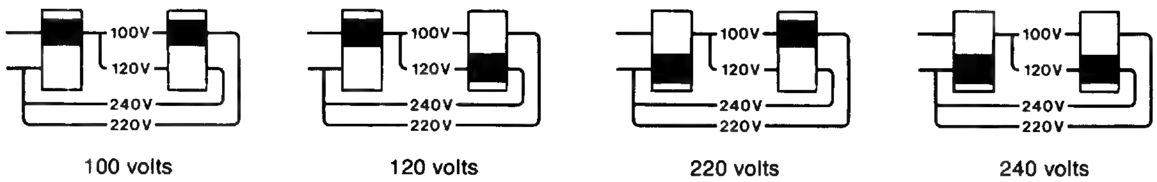
To change a fuse –

- Insert a screwdriver or a coin in the slot of the fuse cap on the rear panel (see page 77).
- Press in slightly on the cap and turn it counterclockwise.
- Pull the fuse cap from the rear panel.
- Remove the original fuse from the fuse cap and install the new fuse (either end) in the cap.
- Install the fuse cap and fuse on the rear panel. Press in slightly on the cap and turn it clockwise.

A

Power Requirements

The 9885M or S can operate on line voltage of either 100, 120, 220, or 240 Vac (+5%, −10%). The line frequency must be within 3.5% of 50 or 60 Hz. The voltage selector switches on the rear panel must be set to the nominal ac line voltage in your area. The next illustration shows the correct settings for each nominal line voltage.



Switch Settings for the Nominal Powerline Voltages

CAUTION

ALWAYS DISCONNECT THE DRIVE FROM ANY AC POWER SOURCE BEFORE SETTING THE VOLTAGE SELECTOR SWITCHES.

To alter the setting of the selector switches –

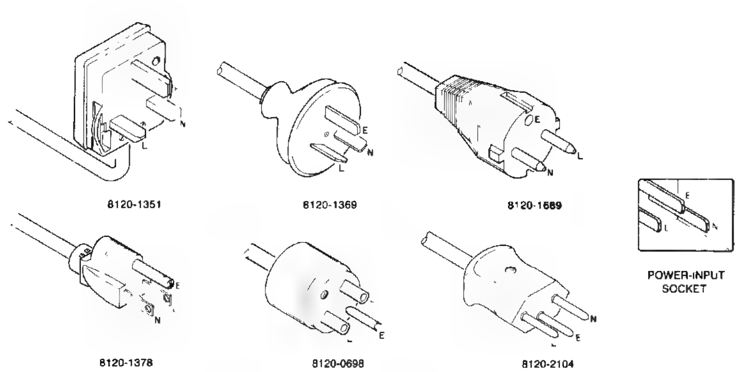
- Insert the top of a small screwdriver (or any small tool) into the slot on the switch.
- Slide the switch so that the position of the slot corresponds to the appropriate voltage as shown.

Option 001 for 50Hz Operation

This option is installed at the factory. It enables the drive to operate properly on a 50Hz line frequency.

Power Cords

Power cords with different plugs are available for the equipment; the part number of each cord is shown next. Each plug has a ground connector. The cord packaged with the equipment depends upon where the equipment is to be delivered. If your equipment has the wrong power cord for your area, please contact your local HP sales and service office.



Power Cord Options

Power cords supplied by HP have polarities matched to the power-input socket of the equipment –

L=Line or Active Conductor (also called "live" or "hot")

N=Neutral or Identified Conductor

E=Earth or Safety Ground

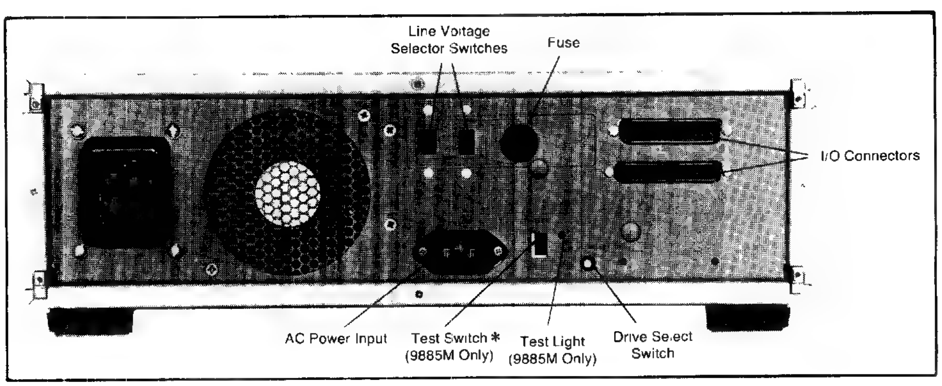
WARNING

IF IT IS NECESSARY TO REPLACE THE POWER CORD, THE REPLACEMENT CORD MUST HAVE THE SAME POLARITY AS THE ORIGINAL. OTHERWISE A SAFETY HAZARD FROM ELECTRICAL SHOCK TO PERSONNEL, WHICH COULD RESULT IN INJURY OR DEATH, MIGHT EXIST. IN ADDITION, THE EQUIPMENT COULD BE SEVERELY DAMAGED IF EVEN A RELATIVELY MINOR INTERNAL FAILURE OCCURRED.

A

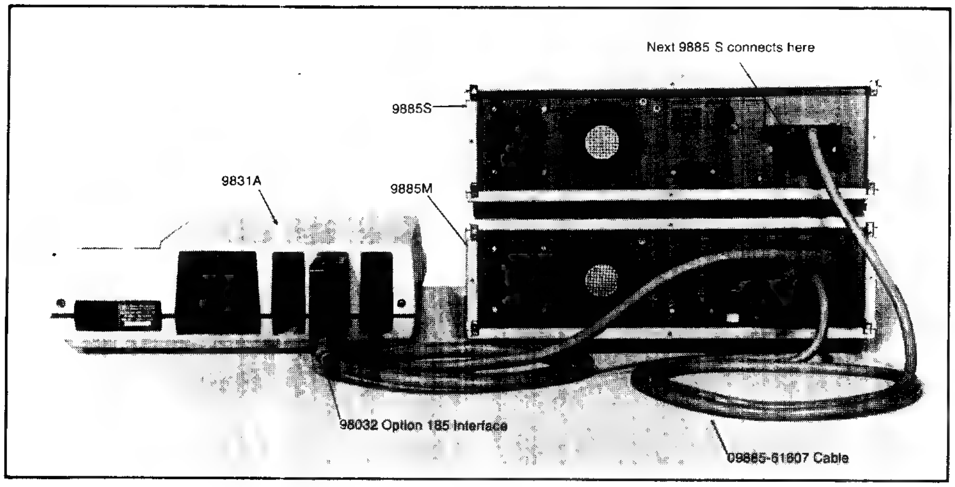
Connecting the System

For a single drive system, connect the 9885M to the desktop computer by inserting the card end of the 98032A Interface into the back of the computer. Connect the other end of the interface to the top I/O connector on the back of the 9885M.



9885 Rear Panel

For multiple drive systems, connect the 9885M as just described. Then, up to three 9885S drives can be connected in series to the 9885M drive by using 09885-61607 cables between drives. (Note: A 9885S drive cannot be connected directly to the desktop computer.



Connecting the 9885 Drives

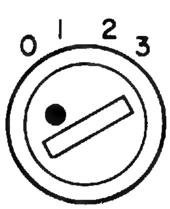
Repeat this procedure for systems with more than one 9885M drive.

Connect one end of the ac power cord to the power-input connector on the rear panel of the desktop computer and the other end to an appropriate ac power source.

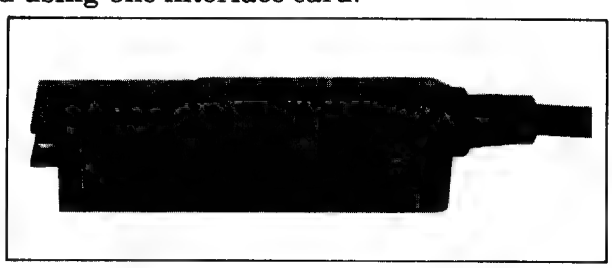
Connect one end of an ac power cord to the power-input connector on the rear panel of each drive and the other end to an appropriate ac power source.

Setting Drive and Select Code Switches

Once all drives are connected, set the drive select switch on the rear panel of each drive to the desired number (0 thru 3). One drive should be set to 0. The drive number selected is the one opposite the dot on the switch. Each of the drives connected to the desktop computer via the same interface card must have a different drive number. A maximum of four drives can be connected using one interface card.



Drive Select Switch



Select Code Switch

Set each 98032A Option 185 Interface Card in your system to a different select code (8 thru 15). One interface should be set to select code 8. Up to eight[1] 9885M drives can be connected to the desktop computer, each having a different select code.

Installing the ROM

Be sure that the desktop computer is switched off before installing the Flexible Disk ROM. With the label right side up, slide the ROM through the ROM slot door. Press it in until the front of the ROM card is even with the front of the machine, as shown.

Installing the Disk

Follow the steps below to install a disk in your drive.

CAUTION

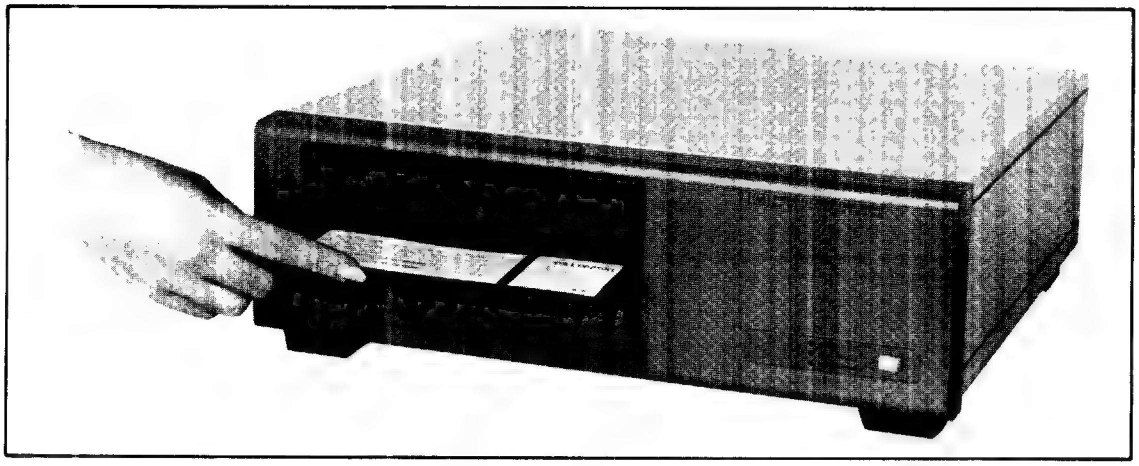
USE ONLY FLEXIBLE DISKS APPROVED BY HP. ANY OTHER DISK MAY CAUSE PERMANENT DAMAGE TO THE READ/WRITE HEAD IN THE DRIVE. FOR A LIST OF APPROVED DISKS, CONTACT AN HP SALES AND SERVICE OFFICE.

A

- Once all drives are properly connected, open the door of the drive by pushing in on the small bar on the front of the drive (below the door handle).
- Then remove the disk from its protective envelope[2] and carefully slide the disk in (label side up and nearest you) until you hear a click.
- Close the door by pressing down firmly on the handle until the door locks closed. (The disk can be installed with power on and the spindle rotating without damage to the disk).
- The disk can be removed by pressing the bar below the handle on the front of the drive. The door springs open and the disk is released. When the disk is removed, it should always be replaced in its protective envelope.

1 An HP 9878A I/O Expander is required if more than three interface cards (including those for 9885M Drives) are connected to the desktop computer.

2 Never remove the disk from its sealed black jacket. For more information about handling disks, see Disk Care Guidelines.



Installing a Disk

Turn On

Once the system is properly connected and the Flexible Disk ROM and a disk are installed, your system is ready to turn on –

- Turn on the desktop computer using the power switch on the right.
- Turn on all 9885M and 9885S power switches located on the front panel of each drive. All drives in a system must be turned on before the system can be operated.

System Tests

A

Programs for testing each 9885 Disk Drive are on the 9831A System Test Cartridge. Instructions for running the tests are in the System Test Booklet.

Self Test

To check the electrical performance of the drive, follow this self test procedure. The drive can be checked with or without a disk installed. The disk door must be closed before the self test can be performed, even if a disk is not installed.

CAUTION

PERFORMING THE SELF TEST WITH A DISK INSTALLED WILL ERASE DATA AND INITIALIZATION ON THE DISK. USE A BLANK (NON-INITIALIZED) DISK FOR THE SELF TEST. (IF THE DISK CONTAINS DATA OR INITIALIZATION AND IS TO BE USED LATER, IT MUST BE COMPLETELY REINITIALIZED.)

To perform the self test –

- Disconnect the 98032A Option 185 Interface cable from the 9885M.
- Close the doors on all the drives in the system.
- Insert the blade of a screwdriver into the slot of the TEST switch on the rear panel of the 9885M and slide the switch down; then release it. The light next to the TEST switch should go on.

WITHOUT the disk installed, the self test –

- Checks the microprocessor and program memory.
- Checks the drive control and drive status circuits.
- Checks the I/O functions.

WITH the disk installed, the self test –

- Checks the microprocessor and program memory.
- Checks the drive control and drive status circuits.
- Checks the I/O functions.
- Checks the read/write electronics.
- Checks the head positioning circuits.

Although the self test does not check all of the drive functions, it gives a high confidence level that the drive is functioning properly. The test takes less than one minute to complete. When the test is complete, the light by the TEST switch will go out.

If the test fails, the light will remain on. So if the light stays on longer than one minute, the test has failed. To repeat the test, be sure all drive doors are closed properly, and the 98032A Interface is disconnected from the drive. Then slide the TEST switch again. If the test fails again, contact your HP sales and service office for assistance.

Maintenance Agreements

Service is an important factor when you buy Hewlett-Packard equipment. If you are to get maximum use from your equipment, it must be in good working order. An HP Maintenance Agreement is the best way to keep your equipment in optimum running condition.

Consider these important advantages –

- Fixed Cost – The cost is the same regardless of the number of calls, so it is a figure that you can budget.
- Priority Service – Your Maintenance Agreement assures that you receive priority treatment, within an agreed upon response time.
- On-Site Service – There is no need to package your equipment and return it to HP. Fast and efficient modular replacement at your location saves you both time and money.
- A Complete Package – A single charge covers labor, parts, and transportation.
- Regular Maintenance – Periodic visits are included, per factory recommendations, to keep your equipment in optimum operating condition.
- Individualized Agreements – Each Maintenance Agreement is tailored to support your equipment configuration and your requirements.

After considering these advantages, we are sure you will see that a Maintenance Agreement is an important and cost-effective investment.

For more information, please contact your local HP sales and service office.

Disk Care Guidelines

The flexible disk is basically maintenance free, but should be handled with care. Here are some guidelines to avoid loss of data or damage to your disks. By following these suggestions, you'll greatly improve the reliability of your disks.

- Use only HP approved disks since use of others can result in damage to your drive. (Contact your local HP sales and service office for a list of recommended manufacturers.)

- Replace worn disk envelopes and always return disks to their storage envelopes after removing them from the drive to protect them from damage. Envelopes can be ordered from HP.
- Since fingerprints on the disk can cause loss of data, **NEVER** touch the surface of the disk showing through the protective sealed jacket.
- Avoid writing on the sealed plastic jacket with lead pencil or ball-point pen. Use a soft felt-tip pen and write on the label only.
- Although the disk is flexible, do not bend or fold it since this, too, can cause damage to the disk.
- Never subject disks to temperatures below 10°C (50°F) or above 52°C (125°F) or relative humdity in excess of 20% to 80%.
- Contamination from dust, ashes, smoke, etc. can damage disks.
- Avoid placing disks in strong magnetic fields like transformers or magnets, since this can cause loss of data.
- Never remove disks from their sealed protective jackets.
- The inside surface of the sealed protective jacket is coated with a special material that cleans the disk as it rotates. Any other method of cleaning may scratch the disk and cause loss of data.

System Reliability

The reliability of your system depends directly on the care you exercise in handling your disks and in avoiding the situations just described. Disks and drives that are not subjected to these “extremes” will perform maintenance free for a longer period of time than those handled without regard to the disk care guidelines.

Appendix B
Reference Information

Disk Specifications

Disk Capacity

The following is a list of the number of words of storage required to store full-precision data elements and string variables. Strings and numerics can be mixed within a record.

Total usable storage per disk – 249,600 words or 1950 records.

Maximum number of files per disk – 352 files

Full-precision numbers per disk – 62,400 numbers

String characters per disk – 491,400 characters

Words per record – 128 words

Disk Speed

Disk speed – 360 revolutions per minute

Average access time – 267 ms

Maximum transfer rate – 11,500 words per second

Instantaneous transfer rate – 31,250 words per second

Storage Requirements

This section describes the space required to store programs and data on the disk.

Program Files

The SAVE statement (Chapter 2) and the DCOPY statement (Chapter 5) each automatically opens a program file large enough to accomodate the program in memory. If the number of words the program uses is not a multiple of 128 (it rarely is), the disk drive rounds the number of physical records reserved in the file to the next whole number. So a program of 127 words requires one physical record, but a program of 129 words requires two physical records.

Data Files

The following table lists the number of words of memory required to store full, split and integer-precision data elements and string variables. Strings and numbers can be mixed within a physical record, as long as each item fits within the bounds of the record.

Data Storage Requirements

Type of Data	Words Per Data Item	Data Items Per Physical Record
full precision	4	32
split precision	2	64
integer precision	2	64
string variable	(see below)	252 characters
EOR or EOF mark[1]	1	-

Notice that the disk drive reserves two words of memory for interger-precision accuracy, rather than one word, as reserved in the desktop computer.

The type of data retrieved, as determined by READ# and MAT READ# statements, can differ from the type of data stored, as determined by PRINT# and MAT PRINT# statements. Data stored with full-precision accuracy, for example, can be retrieved with full, split, or integer precision. An error message is displayed if you try to convert a full- or split-precision number greater than 32,767 to an integer-precision number, or if you try to convert a number greater than 9.999999E63 to split precision.

B

Use this equation to find the space required to store string variables —

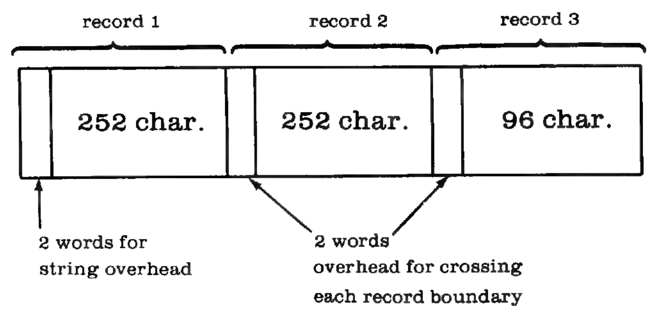
$$\text{No. of words} = \text{INT}\left(\frac{\text{no. of characters} * 2+1}{2}\right) + 2 \text{ per record}$$

1 An extra word is not required for an EOR mark when the record is completely filled, or an EOF mark when the end of file is reached.

As shown, a string requires 2 words of overhead in addition to the normal ½ word per character requirement (plus another ½ word if the string has an odd number of characters). So up to 252 characters (and 2 words of overhead) can be stored into each record. Printing longer strings will overlap record boundaries. Two extra words of overhead are needed, however, when the string overlaps each boundary.

For example, storing a 600-character string requires –

- 300 words for string characters
- 2 words of string overhead
- 2 words of extra overhead as the string crosses a record boundary (4 words are needed here).



Data Verification

Write Verification

The disk drive routinely checks to ensure that all the information being transferred between it and the desktop computer's memory corresponds exactly to the original. During PRINT# and MAT PRINT# operations, this is done by automatically executing the entire data list a second time and then comparing the two copies; if a difference is found, the operation is repeated up to 10 times in an attempt to store the information correctly. If the write operation is not successful after 10 tries, it's canceled and ERROR 59 is displayed. (ERROR 59 also indicates a tape verify error, as described at the back of Chapter 6 of the 9831A Operating and Programming Manual.)

If you get ERROR 59 during a PRINT# or MAT PRINT# operation, it probably means that the surface of the disk where the data is to be written has been damaged.[1] Since other write errors will probably occur with that disk, you should initialize a new disk and transfer any data on the damaged one to the new one (use DCOPY if you have more than one disk drive). Then you can reinitialize the damaged disk (see Appendix C) and test it, as described in the 9831A System Test Booklet. The test will tell you if the disk is permanently damaged.

1 For auto verification, the entire list of data items in each PRINT# statement is executed a second time (immediately after the data is written on the disk) so that each item can be compared with its corresponding item on the disk. Since the entire data list is re-executed a second time, **ERROR 59** will occur when an item in the data list generates a different value each time it is executed. For example, **PRINT#;RNDO** computes a different random number each time **RNDO** is executed. When this situation can't be avoided, switch auto-verify OFF.

B

If you wish to switch off the auto-verify routine, use the VERIFY statement –

`VERIFY OFF` turns off the auto-verify routine.

`VERIFY [ON]` turns auto-verify back on.

Since each write operation takes at least twice as long to execute with the auto-verify routine, it may be helpful to switch auto-verify off when many write operations are to be done in quick succession – but remember that the data will not be automatically checked when auto-verify is switched off.

The VERIFY statement also controls data verification with tape cartridges, as described in Chapter 6 of the 9831A Operating and Programming Manual.

Read Verification

During a read (load) operation, the disk drive automatically compares a checkword with another checkword originally written on the disk; if a difference is found, the data is reread again. If the data cannot be read successfully after 10 tries, the operation is canceled with ERROR 85.

If you get ERROR 85, a read error, data cannot be read from a record because the checkword for that record is not identical to the one generated during the read operation. For data files, the DCOPY statement can be used to correct the checkword by recopying the entire file back into the same space on the disk. This operation will not halt when reading the file, but ERROR 85 will be displayed after the file is rewritten with a new, correct checkword. For example, to recopy file "data" on drive number 1, execute –

B

```
DCOPY "data", 1 TO "data", 1
```

If ERROR 85 occurs while reading a program file, however, DCOPY cannot be used to correct the checkword. In this case, the program is not recoverable.

Glossary of Disk Terms

availability table – Table in systems area that monitors the amount and location of remaining disk space.

backup track – Track 1 of an initialized disk contains the same information as track 0: the systems table, the file directory and the availability table.

checkword – A unique 16-bit word automatically written on the disk at the end of each record during a write operation. Also called the CRC –Cyclic Redundancy Code.

checkword error – When a record is read, a checkword is generated and is compared to the checkword at the end of the record for data validity. If not identical after 9 rereads, ERROR 85 is displayed.

controller – An electronic assembly in the 9885M drive (not contained in the 9885S) that monitors and controls all drive functions.

defective track – A track on the disk where the reading and writing of data is not possible, usually because of a scratch, dirt, or lack of magnetic oxide on the surface of the disk. The number of defective tracks is identified during initialization and is recorded in the systems table.

disk – The flexible disk is the storage medium for the 9885M or 9885S drive. Data is written on a thin magnetic oxide film coated on plastic. The disk is enclosed in a sealed plastic jacket for protection.

drive – The 9885M and 9885S are referred to as drives.

drive number – The drive number (0 thru 3) is selected by the drive select switch on the rear panel of the disk drive.

double density – The type of recording techniques used by the 9885, giving increased storage capacity and higher transfer rates over a tape cartridge.

EOF mark – A mark placed in the first word of each record when a file is opened and at the end of the data in a file when the END parameter is used. Data cannot be read past this mark, although it can be written.

EOR mark – Marks placed after the last data item when the END parameter is omitted.

file – A file is one or more user records written on the disk.

file directory – A directory in the systems area containing entries for every file on the disk indicating file name, size, type and location.

flexible disk – The disk is also referred to as a flexible disk.

head – The read/write head contains the read, write, and erase elements (coils) encased in ceramic. The head is in contact with the lower surface of the disk when data is transferred.

header – A unique bit pattern representing the address of the record, written at the beginning of each record during initialization.

hard error – A hard error is usually non-recoverable. A softwar error-recovery routine, however, can be used to try to recover from the error.

initialize – When a disk is initialized, addresses are written on it, it is tested by writing and reading patterns from the disk, and the systems area is set up.

label – An alphanumeric name assigned to each disk by using PRINT LABEL.

load pad – Pad opposite the head (touching the upper surface of the disk).

logical file access – A method of storing and retrieving data items separately, word by word.

multirecord strings – Long string variables are automatically stored in successive records.

random file access – A method of storing and retrieving data items, record by record.

record – A block of 128 data words written on the disk, following a header and followed by a checkword.

B

seek – Movement of the head from one track to another.

soft error – Soft errors are recoverable and are usually caused by dirt in the air or on the disk, random electrical noise, small defects on the disk, or a defective load pad.

select code – The address which each interface card and its peripheral devices responds to. In a 9885 system, each 98032A Option 185 Interface must be set to a different select code (8 thru 15).

serial file access – A method of storing and retrieving data items as a group, instead of individually.

storage area – Tracks 2 thru 66 are available for your data storage.

systems area – The systems area consists of disk tracks 0 and 1. It contains the systems table, file directory, availability table, and backup track.

systems table – Table in systems area indicating the computer used to initialize the disk, number of defective tracks, beginning of user area, and an optional disk label (name).

track – Any one of 67 concentric circles on the surface of the disk, about 0.012 inches wide and numbered 0 thru 66.

transition – A flux reversal caused by writing on the disk producing an electrical transition during a read that is decoded into bits (0 or 1).

tight margin – A restriction in the time allowed for a read during which a flux transition can be interpreted as a bit (a 1 or 0).

B

verify error – When auto-verification (VERIFY) if on, each PRINT# or MAT PRINT# operation is reread under tight margin to ensure accuracy. If an error is found during a write operation, ERROR 59 is displayed.

word – The smallest unit of disk storage and equivalent to a word of read/write memory. A record can hold 128 words of data or program information.

WRITE tab – An opaque tab which permits writing on the disk. When the WRITE tab is removed, writing on the disk is not allowed.

ASCII Character Codes

ASCII Char.	EQUIVALENT FORMS			ASCII Char.	EQUIVALENT FORMS			ASCII Char.	EQUIVALENT FORMS			ASCII Char.	EQUIVALENT FORMS		
	Binary	Octal	Dec		Binary	Octal	Dec		Binary	Octal	Dec		Binary	Octal	Dec
NULL	00000000	000	0	space	00100000	040	32	@	01000000	100	64	`	01100000	140	96
SOH	00000001	001	1	!	00100001	041	33	A	01000001	101	65	a	01100001	141	97
STX	00000010	002	2	"	00100010	042	34	B	01000010	102	66	b	01100010	142	98
ETX	00000011	003	3	#	00100011	043	35	C	01000011	103	67	c	01100011	143	99
EOT	00000100	004	4	$	00100100	044	36	D	01000100	104	68	d	01100100	144	100
ENQ	00000101	005	5	%	00100101	045	37	E	01000101	105	69	e	01100101	145	101
ACK	00000110	006	6	&	00100110	046	38	F	01000110	106	70	f	01100110	146	102
BELL	00000111	007	7		00100111	047	39	G	01000111	107	71	g	01100111	147	103
BS	00001000	010	8	(	00101000	050	40	H	01001000	110	72	h	01101000	150	104
HT	00001001	011	9	)	00101001	051	41	I	01001001	111	73	i	01101001	151	105
LF	00001010	012	10	*	00101010	052	42	J	01001010	112	74	j	01101010	152	106
V_{TAB}	00001011	013	11	+	00101011	053	43	K	01001011	113	75	k	01101011	153	107
FF	00001100	014	12	,	00101100	054	44	L	01001100	114	76	l	01101100	154	108
CR	00001101	015	13		00101101	055	45	M	01001101	115	77	m	01101101	155	109
SO	00001110	016	14	.	00101110	056	46	N	01001110	116	78	n	01101110	156	110
SI	00001111	017	15	/	00101111	057	47	O	01001111	117	79	o	01101111	157	111
DLE	00010000	020	16	Ø	00110000	060	48	P	01010000	120	80	p	01110000	160	112
DC_1	00010001	021	17	1	00110001	061	49	Q	01010001	121	81	q	01110001	161	113
DC_2	00010010	022	18	2	00110010	062	50	R	01010010	122	82	r	01110010	162	114
DC_3	00010011	023	19	3	00110011	063	51	S	01010011	123	83	s	01110011	163	115
DC_4	00010100	024	20	4	00110100	064	52	T	01010100	124	84	t	01110100	164	116
NAK	00010101	025	21	5	00110101	065	53	U	01010101	125	85	u	01110101	165	117
SYNC	00010110	026	22	6	00110110	066	54	V	01010110	126	86	v	01110110	166	118
ETB	00010111	027	23	7	00110111	067	55	W	01010111	127	87	w	01110111	167	119
CAN	00011000	030	24	8	00111000	070	56	X	01011000	130	88	x	01111000	170	120
EM	00011001	031	25	9	00111001	071	57	Y	01011001	131	89	y	01111001	171	121
SUB	00011010	032	26	:	00111010	072	58	Z	01011010	132	90	z	01111010	172	122
ESC	00011011	033	27	;	00111011	073	59	[	01011011	133	91	{	01111011	173	123
FS	00011100	034	28	<	00111100	074	60	\	01011100	134	92	¦	01111100	174	124
GS	00011101	035	29	=	00111101	075	61	]	01011101	135	93	}	01111101	175	125
RS	00011110	036	30	>	00111110	076	62	^	01011110	136	94	~	01111110	176	126
US	00011111	037	31	?	00111111	077	63	_	01011111	137	95	DEL	01111111	177	127

Appendix C
Disk Utility Routines

The Disk Utility Routines are a set of binary and BASIC language programs that perform supplementary disk operations. These binary statements are available –

INIT	Initialize a blank disk with a systems area.
KILLALL	Erases all files from a disk.
BACKUP	Duplicates all files from one disk to another.
REPACK	Packs all files into one area, freeing up unused disk space.
DISKDUMP	Transfer information from one disk to tape cartridges.
DFDUMP	Transfers information from one data file to tape.
DISKLOAD	Transfers information from tape cartridges to disks.
DFLOAD	Transfers information from a tape file to a disk data file.

These commands are available for recovering data and performing other useful operations –

HELP	Produces a list of disk utility commands.
SCAT	Produces a catalog listing using the spare directory.
DCOMPARE	Compares the spare and main directories.
EXCHANGE	Exchanges the main the spare directories.
MAIN TO SPARE	Copies the main directory into the spare.
SPARE TO MAIN	Copies the spare directory into the main.
LIST AT	Produces a list of the availability table.

(continued)

RECREATE AT	Recreates the availability table from the directory.
TRDUMP	Dumps the specified track to tape.
TRLOAD	Loads the track recorded on tape back onto the disk.
TRINIT	Initializes the specified track.
DFLIST	Lists contents of records in data files.
DFEDIT	Allows editing specified records in a data file.
CHECK	Checks format of record 0 of System Table.

CAUTION

MOST OF THE ROUTINES AND COMMANDS DESCRIBED IN THIS APPENDIX ARE INTENDED FOR THE ADVANCED PROGRAMMER. IMPROPER USE CAN QUICKLY ERASE ALL INFORMATION ON THE DISK!

The routines described here are available on the Utility Routines Disk (Part No. 09885-90014), which is supplied with the 9885M (option 031) Disk Drive.

Most of the routines assume that the Utility Routines Disk is used in a drive set to drive 0 and select code 8. If not, execute the UNIT statement to specify the drive being used (UNIT is covered in Chapter 5).

If a second drive is available, the general procedures can be simplified by using the Utility Routines Disk in the master drive (drive 0, select code 8) and the second disk in the slave drive.

Initializing Blank Disks

The initialization (INIT) routine writes addresses on the disk so that specific locations may be referenced by the system. During initialization, test patterns are also written on the disk and then read for verification. This takes less than four minutes per disk. Once initialization starts, all previous information on the disk is lost.

C

Each blank disk must be initialized before it can be used with your system. Once this procedure is complete, the disk remains initialized and does not have to be reinitialized each tme the system is turned on. Remember that a disk can be accidentally erased via the Pattern Test or the 9885 Self Test, as described in the 9831A System Test Booklet.

Type in the drive number holding the new disk.

6. Now verify the drive numbers –

   `FROM=nn, TO=nn? HIT CONTINUE IF OK`

   If the drive numbers are correct, press (CONT). If not, press (STOP) to abort the routine.

   ERROR 77 indicates that the select code specified by the routine is not currently set. If this error occurs, use the UNIT statement to specify the correct drive select code. Then return to Step 4.

7. The routine now duplicates all files from the original disk onto the new one. The display is –

   `BACKING-UP DISKETTE`

The time required for the routine varies (from 2 thru 12 minutes) depending on how much read/write memory is available and whether VERIFY is ON or OFF. If verify is ON, the information on the new disk is compared against that on the original for accuracy. ERROR 59 will result if the VERIFY comparison fails.

The routine automatically compares the number of available tracks on each disk before duplicating the files. If the new disk has less defective tracks than the original disk, the duplication is performed. If the new disk has more defective tracks than the original, however, the routine checks to see if the defective tracks are needed. If not, the duplication is performed. If a REPACK is required (to free any extra storage space on the original disk), ERROR 300 is displayed. See The REPACK Routine next. If there are too many defective tracks on the new disk, ERROR 301 indicates that the routine has been aborted.

When the new disk has a different number of defective tracks than the original one (but duplication is still possible), the availability table on the new disk is updated to reflect the actual amount of unused space on that disk.

C

The REPACK Routine

The REPACK routine moves all user files to the beginning of the storage area on a disk. The routine consolidates all unused space (from files previously killed) into one area, making future use of the disk more efficient. Execution of disk statements is often faster, since the average distance between user files is decreased.

This routine can take up to 12 minutes, depending on how much read/write memory is available. For fastest results, execute `ERASE A` before continuing.

1. Insert the Utility Routines Disk in the drive and close the door.
2. Key in and execute – `GET BIN "REPACK"`
3. Replace the Utility Routines Disk with the disk to be repacked and close the door.
4. Key in and execute – `REPACK` n where n is the disk drive number.
5. The routine first takes about 10 seconds to compact the file directory –

   ```
   COMPACTING DIRECTORY
   ```

6. Then the storage area is repacked –

   ```
   REPACKING DISKETTE
   ```

   The running time depends upon the amount of available read/write memory, the number of files to be repacked, and whether VERIFY is ON or OFF (up to 12 minutes). The file directory is updated after each file is repacked.

If a read or write error occurs while the directory is being compacted, the routine is aborted. A read error does not affect the contents of the directory, but indicates that the main directory (track 0) may be partially defective. A write error, however, indicates that either directory is bad and some files may be inaccessible. In this case, execute `CAT` to list the files available.

If a read or write error occurs while the storage area is being reproduced, an error message indicates the defective file by name. The routine will then either halt or display `RECOVERING FILE` and halt after a few moments. This last display appears while the routine is rearranging the files so that they are accessible and updating the availability table.

The Dump and Load Routines

The disk dump routines store the entire disk or indicated data file from the disk to the specified tape files. The disk load routines transfer data from tape files created by the disk dump routine to disk files.

To load the disk dump and load routines, execute –

```
GET BIN "DMPLOD"
```

The DISK DUMP Routine

DISK DUMP [no. of disk records per tape file]

The DISK DUMP routine transfers the entire disk onto tape cartridges, starting with track 0, file 0 of the first tape cartridge.

The optional parameter indicates whether the tape is to be marked automatically, or not, and the number of disk records to be dumped per tape file. The parameter can be an integer other than 0 (128 words of memory are needed per record). If sufficient memory is not available to perform the dump (or load), ERROR 2 occurs. If the parameter is positive, the tape is marked automatically. If negative, you must have marked the tapes beforehand.

The File Dump Routine

DFDUMP disk file name[, tape track[, tape file [, no. of records per file]]]

This routine transfers the indicated data file from the disk to tape, starting at the specified tape file. The optional parameter has the same effect here as for the DISK DUMP.

The DFDUMP routine transfers disk information on the current track (starting at the first file number given) until the current tape track is filled (or the null file is reached for premarked tapes). After the track is filled (if the current track is 0), the tape automatically rewinds and continues with track 1, file 0. If the current track is track 1, however, NEXT TAPE, PRESS CONTINUE is displayed and the routine waits for tapes to be changed; then it continues with track 0, file 0 of the new tape.

If SET RECORD, PRESS CONTINUE is displayed, pull out the cartridge, slide the RECORD tab to the right, reinsert the cartridge, and press CONT.

The DISK LOAD Routine

The DISK LOAD routine transfers the entire disk from the tape files recorded via DISK DUMP.

DISK LOAD

The disk is loaded, starting from tape track 0, record 0. If the DISK DUMP used ten records per tape file, for example, there must be 1280 words of available memory to perform the load. If not, ERROR 2 results. With one record per tape file, there are no memory requirements.

C

The FILE LOAD Routine

`DFLOAD` disk file name[, tape track[, tape file]]

This routine transfers data, starting from a specified tape file on the current track into the disk data file named. The tape file must have been created by a DFDUMP routine.

The data file is transferred, starting at the specified tape file number on the current track. The tape file number must be the same as for the corresponding DFDUMP. If the DFDUMP used ten records per tape file, for example, there must be 1280 words of available memory to perform the DFLOAD. If not, `ERROR 2` results. With one record per tape file, there are no memory requirements. The size of the disk file must be greater than or equal to the size of the original file from which the data was dumped, otherwise `ERROR 93` is displayed. Any extra records in the destination file are not affected.

NOTE

The dump and load statements require that the Flexible Disk ROM be installed. ERROR 300 will occur if you attempt to execute or store these statements without the ROM.

Utility Routines Commands

The utility routines commands are a set of BASIC language programs accessed by first loading the "UTILTY" binary program from the Utility Routines Disk –

```
GET BIN "UTILTY"
```

Then executing each of the command names loads and runs the corresponding BASIC language program from the disk.

Since these programs are stored on the Utility Routines Disk, it must be installed whenever one of these commands is executed. This poses a problem on single disk systems, since it's not possible to have both the utility disk and the user's disk installed simultaneously. When only one disk drive is available, the operator is asked `PROPER DISKETTE INSTALLED?` after executing each command, except HELP, to remind the operator to remove the utility disk and install the user's disk. Then press (Y) to continue command execution. Be sure to replace the utility disk when the command is complete.

The HELP Command

HELP

This command produces a list of all 14 commands along with a brief description of each. The listing is printed on the standard printer. Here's a sample printout –

```
LIST OF AVAILABLE COMMANDS:

HELP           Produces a list of all available commands.
SCAT           Produces a catalog listing using the spare directory.
DCOMPARE       Compares the spare and main directories.
EXCHANGE       Exchanges the main and spare directories.
MAIN TO SPARE  Copies the main directory into the spare.
SPARE TO MAIN  Copies the spare directory into the main.
LIST AT        Produces a list of the availability table.
RECREATE AT    Recreates the availability table from the directory.
TRDUMP         Dumps the specified track to tape.
TRLOAD         Loads the track recorded on tape back onto the diskette.
TRINIT         Initializes the specified track.
DFLIST         Lists contents of records in data files.
DFEDIT         Allows editing specified records in a data file.
CHECK          Checks format of record 0 of system table.
```

The SCAT Command

SCAT

The SCAT (spare catalog) command produces a catalog listing using the spare directory rather than the main. Note that since parameters are not permitted, SCAT works only for the drive specified by the most recent UNIT statement. Here's a sample listing –

```
   SPARE CATALOG OF DRIVE 1  (SC=8)
         AVAILABLE RECORDS:  1846
NAME     TYPE    LENGTH    TRACK    RECORD
-------  ------  -------   -----    ------

SCAT     PROG      779W      2        0
HELP     PROG      579W      2        7
SPTOMN   PROG      412W      5        4
MNTOSP   PROG      371W      2       15
EXCHAN   PROG      439W      2       18
UTILTY   BINARY   1537W      2       22
RECRAT   PROG      700W      3        5
CHECKR   PROG      562W      3       11
DFLIST   PROG     1514W      3       16
DFEDIT   PROG     1395W      3       29
COMPAR   PROG      414W      4       11
LISTAT   PROG      445W      4       15
TRDUMP   PROG      373W      2       12
TRLOAD   PROG      263W      4       23
TRINIT   PROG      900W      4       26
DMPLOD   BINARY    512W      4       19
REPACK   BINARY   1024W      5        8
```

C

The DCOMPARE Command

DCOMPARE

This command compares the spare directory to the main. Any differences are noted on the standard printer. If differences occur, the following actions should be taken –

Availability Table in error – Use RECREATE AT to build a new table. Then execute DCOMPARE again.

Directory in error – Use CAT and SCAT to determine which directory is incorrect. Then copy the good one to the bad one.

Record 0 of System Table is in error – Use CHECK to list the contents of record 0 of the main system table. If it's OK, copy MAIN TO SPARE. If not, try an EXCHANGE, and then use CHECK to check the spare table. Then use CAT to check the catalog. If the catalog is correct, copy MAIN TO SPARE. If not, use EXCHANGE again to put the disk back in its original configuration. As a last try, dump track 0 onto tape by using TRDUMP. Then initialize the track using TRINIT and let the system copy the spare to the main. If the catalog (CAT) is still wrong, save the tape containing track 0 and call HP for assistance.

The EXCHANGE Command

EXCHANGE

This command swaps the main the spare directories. Except as an emergency recovery procedure, EXCHANGE should not be used if DCOMPARE indicates that record 0 of the system table doesn't match the spare. See the DCOMPARE command.

The MAIN TO SPARE Command

MAIN TO SPARE

C

This command copies the main directory to the spare.

The SPARE TO MAIN Command

SPARE TO MAIN

This command copies the spare directory to the main.

The LIST AT Command

```
LIST AT
```

This command produces a list of the availability table. The unused storage areas on the disk, their locations and sizes are listed. The amount of unused storage area is then printed. For example –

```
AVAILABILITY TABLE:

TRACK #   RECORD #    LENGTH
===== =   ====== =    ======

      2         22         4
      9          1      1739
      2          9         8
      8         22         1
                   ---------
UNUSED STORAGE:         1752
```

The RECREATE AT Command

```
RECREATE AT
```

This command recreates the availability table based on the information in the directory. RECREATE AT should be used when the directory is known to be correct, but either the main and spare availability tables don't match or the availability table does not indicate the space actually available on the disk.

C

The TRDUMP Command

```
TRDUMP
```

The TRDUMP (track dump) command records the contents of the specified track onto a tape. The tape is rewound and three files are marked on both tracks 0 and 1 to provide duplication of information in case a tape error occurs.

The TRLOAD Command

```
TRLOAD
```

The TRLOAD (track load) command reloads the track previously dumped on tape back onto the disk. Since the track number is retained in the data recorded on tape, it's not necessary to re-enter the track number to execute TRLOAD.

The TRINIT Command

```
TRINIT
```

The TRINIT (track initialize) command re-initializes the specified track. Before executing this command, the track should first be dumped to tape using `TRDUMP`, and afterwards reloaded using `TRLOAD`. If the track to be initialized is in either the spare directory or the main directory, the operator is asked if the main is to be reloaded from the spare, or vice versa, as appropriate. This is not necessary, but may be a desirable precaution. If the track to be initialized is either the main or spare directory, however, DCOMPARE should be executed afterwards to verify that the contents are correct.

TRINIT should be used whenever ERROR 83, 84, or 85 occurs while reading information from the same track. To determine which track is bad, use the Checkread (CKRD) test, as described in the 9831A System Test Booklet. This test is also recorded on the Utility Routines Disk and can be loaded into the computer by executing –

```
GET BIN "DSKTST"
```

The DFLIST Command

C

```
DFLIST
```

The DFLIST (data file list) command prints a list of the specified data file. The user is requested to specify whether the file is a "logical file " (a file created using logical prints), and a range of records to be printed.

For example, here's a list of records 1 thru 5 of a 50-record file named "X" –

```
File X ( 50   records)

List of records 1    thru 5


Record  1

F: 3.1415          S: 13                 I: 14
$:Hi there!

EOF ( 15   words used)


Records 2    thru 5

EOF
```

Full-precision numbers are prefixed by "F:", split by "S:", integer by "I:", and strings by "$:". If a string is too long to completely fit on a line, the remainder is printed on following lines preceded by "&:". "Null string" indicates a string length of 0. If the record terminates with an EOF, "EOF" is printed.

As another example, this listing is of a 1-record file named "Y" –

```
File Y ( 1    record)

List of record 1


Record  1

I: UNDEF           S: UNDEF             F: UNDEF
EOR ( 8    words used)


Physical end of file.
```

C

"UNDEF" indicates undefined data items.

A warning is printed if the first item in the first record to be printed is of mid-string or end-string type (this does not necessarily constitute an error).

In the case of a logical file, the contents (if any) following the first EOR or EOF mark are printed.

The DFEDIT Command

```
DFEDIT
```

The DFEDIT (data file edit) command allows you to review and, if needed, change any item in a data file.

The DFEDIT command should be used to correct data in a file in which that data has been destroyed. Here is the general procedure –

1. After executing this command, you will be asked –

```
FILE NAME?
```

2. Enter the name of the file with the record to be corrected. Then you will be asked –

```
EDIT RECORD #?
```

3. Enter the number of the record which has the error. Then you will be asked –

```
START WORD # (1-128)?
```

4. Enter the word at which editing is to start. If this is not a logical file, the number should be 1; if it's a logical file, the number should be the starting word number of the desired logical record.
5. Then a header specifying the file name, start record and start word for the edit will be printed, along with a title for the data to be printed and the first item in the file (see sample printout). You will be asked –

```
IS ITEM CORRECT?
```

6. If it's correct, press Y and the next item will be printed. (If the last item was an EOR or an EOF, the edit is complete.) If it is not correct, press N –

```
IS TYPE CORRECT?
```

7. If the data type is not correct, press N. You will then be asked for the new type and the new length, if it is a string. The data item will then be printed again (return to step 6). If it's correct, press Y and you will be asked –

```
IS VALUE CORRECT?
```

C

8. If the value is correct, press Y. The next item is printed (return to step 6). If the value is not correct, press N. Then enter the correct value as requested. For long strings, enter substring of 80 characters and press and press EXEC after each, until the entire string is entered. After the value has been changed, the new value will be printed.
9. Return to step 6 to check each item in the record.

```
EDIT OF X
RECORD 1     , WORD 1

ITEM# WORD  TYPE    VALUE
----- ----  ------- -----

    1    1 FULL      3.1415

    2    5 SPLIT     13

    2    5 SPLIT     13.98

    3    7 INTEGER   14

    3    7 SPLIT     8.0014

    3    7 SPLIT     78.3

    4    9 $9       Hi there!

    4    9 $9       123456789

    5   16 EOF
```

Here's another example printout, showing the messages printed when PRINT ALL is switched ON –

```
dfedit
DATA FILE EDIT
FILE NAME?X
EDIT RECORD #?1
START WORD # (1-128)?1

EDIT OF X
RECORD 1     , WORD 1

ITEM# WORD  TYPE    VALUE
----- ----  ------- -----

    1    1 FULL      3.1415

IS ITEM CORRECT?Y
    2    5 SPLIT     13

IS ITEM CORRECT?N
IS TYPE CORRECT?Y
IS VALUE CORRECT?N
NEW VALUE?13.98
    2    5 SPLIT     13.98
```

C

```
IS ITEM CORRECT?Y
    3     7 INTEGER  14

IS ITEM CORRECT?N
IS TYPE CORRECT?N
NEW TYPE CODE?5
    3     7 SPLIT     8.0014

IS ITEM CORRECT?N
IS TYPE CORRECT?Y
IS VALUE CORRECT?N
NEW VALUE?78.3
    3     7 SPLIT     78.3

IS ITEM CORRECT?Y
    4     9 $9       Hi there!

IS ITEM CORRECT?N
IS TYPE CORRECT?Y
IS VALUE CORRECT?N
NEW VALUE (9 CHAR. LEFT)?12345
NEW VALUE (4 CHAR. LEFT)?6789
    4     9 $9       123456789

IS ITEM CORRECT?Y
    5    16 EOF

IS ITEM CORRECT?Y
EDIT COMPLETE.
```

The CHECK Command

```
CHECK
```

This command checks the Systems Table (record 0) to verify that it is correct, and then prints information contained in the table. Here's a sample printout –

```
List of record 0 of system table

Unit= 1     , SC= 8

Initialized by                     9831
Records/track                      30
Total good tracks                  67
Track # of spare                   1
First record of directory          1
First record of AT                 23
First record after AT              29
First user track                   2
Number of user tracks              65
Number of defective tracks         0
```

Appendix D
Disk BASIC Syntax

Syntax Guidelines

The disk operations available with the Flexible Disk ROM are summarized here. The conventions and terms most often used within the syntaxes for statements and functions are –

brackets [ ]	Items enclosed in brackets are optional.
dot matrix	Items in dot matrix must appear as shown.
...	Dots indicate that the preceding items can be repeated.
drive no.	An integer expression from 1 thru 3 indicating which drive should be used. Also, see "The UNIT Statement" in Chapter 5.
file name	The name used to define a specific file. It can contain up to six characters; quotes ("), commas, colons, blank spaces, a leading asterisk (*), and nulls (decimal 0) cannot be used in the file name. The name can be either text (characters within quotes) or a string variable (quotes not used).
1st line number	An integer number referencing a program line.
2nd line number	The 2nd line number can be used only with the 1st line number.
file no.	An integer expression from 1 thru 10 representing a file name, as specified via a FILES or ASSIGN statement.
record no.	An integer expression specifying a physical record within a file.
word pointer	An integer expression from 1 thru 129 specifying the starting point (word) for logical PRINT# and READ# operations.

ASSIGN Statement

ASSIGN file name ; file no. ; return variable [; drive no.]

Assigns a file number to a single file name and optionally, the drive number for the file specified. An optional return variable can be used for further file information.

Return value	Meaning
0	File is available.
1	File type is not data.
2	Drive no. is not from 0 thru 3.
3	File has not been opened.
4	File no. is not from 1 thru 10.

AVAIL Function

AVAIL drive no.

Returns the total number of records available (unused) on the specified file.

CAT Statement

CAT [drive no.[; printer select code]]

Prints information about all user files on the disk. An example printout is shown on page 16.

CERROR Statement

CERROR

Cancels any error recovery routine set by SERROR. The SERROR statement is described in Chapter 3 of the 9831A Operating and Programming Manual.

CHAIN Statement

CHAIN file name [; 1st line number[; 2nd line number]]

D

Loads the program specified from the disk into the memory and retains the values of all variables. (Same line number rules as for the GET statement apply.)

DBYTE Statement

DBYTE expression ; string variable

Converts an expression to a string character and stores the character in the string variable or substring.

DCOPY

DCOPY file name [; drive no.] TO file name [; drive no.]

Duplicates the contents of one file into another. The optional drive numbers allow you to copy a file from one drive to another.

DEXP Statement

DEXP expression ; string variable

Converts the value of the specified expression into a 4-digit character string with leading zeros. A substring may be specified.

DGET Statement

DGET file name [; 0]

Loads a source program previously stored with DSAVE back into the memory. The option 0 parameter prevents the program from being automatically run.

DREN Statement

DREN old file name TO new file name

Changes the name of any file; the contents of the file remain the same.

DSAVE Statement

DSAVE file name [; 1st line no.[; 2nd line no.]]

Stores a program as a **source** program, a series of data strings. The file name must have prviously been opened.

FIL Function

FIL drive no.

Returns the size (in records) of the largest unused space available on the specified disk.

FILES Statement

FILES file name$_1$ or *[: drive no.$_1$][, file name$_2$ or *[: drive no.$_2$]...]

Assigns file numbers (1 thru 10) to the files named and optionally the drive number for each file named. Asterisks may be substituted for file names when an ASSIGN statement follows. File names must be text without quotes; string names are not allowed in FILES.

FRAC Function

FRAC expression

Returns the fractional part of the value of the expression.

GET Statement

GET file name [, 1st line number[, 2nd line number]]

Loads the program specified from the disk into the memory. Variable values are not retained.

1st line number – If specified, the loaded program lines are renumbered with the beginning line number corresponding to the specified first line number. (Program lines in memory with line numbers lower than the first line number are retained.)

2nd line number – Execution starts at the second line number. (If executed from within a program, execution of the program begins automatically.) If the second line number is omitted, program execution begins at the first line number.

D

GET BIN Statement

GET BIN file name

Loads a binary program from the disk into the memory.

GET KEY Statement

`GET KEY` file name

Loads all special function key definitions from the specified file to the special function keys.

GET MEM Statement

`GET MEM` file name

Loads the entire read/write memory from the specified file, returning machine to its state before `SAVE MEM` was executed.

IF END# Statement

`IF END#` file no. `THEN` line no.

Sets up a branching condition in the program to automatically exit a READ# or PRINT# operation when an end of file (EOF) mark is seen.

KILL Statement

`KILL` file name

Erases the file from the disk and makes the file space available. The availbility table is automatically updated (repacked) following KILL.

LEX Function

`LEX (`string variable `,` "text" or string variable `)`

Compares the two strings, character-by-character, according to the ASCII value of each character. If the first string is greater than the second, 1 is returned. If the strings are of equal value, 0 is returned. If the first string is less than the second, −1 is returned.

MAT CON Statement

`MAT CON` expression `;` array list

Initializes each element of each specified array to the value of the expression.

D

MAT PRINT# Statement

MAT PRINT # file no. [, record no.[, word pointer]] ; matrix list

Prints an entire matrix into a specified record or file.

MAT READ# Statement

MAT READ # file no. [, record no.[, word pointer]] ; matrix list

Reads the matrix from a specified record or file.

MAT ZERO Statement

MAT ZERO array list

Sets all elements in the specified array(s) to 0.

NUM Function

NUM (string variable)

Returns the ASCII-decimal value of the first character of the specified string or substring.

OPEN Statement

OPEN file name , number of records

Creates a data file on the disk, with the indicated number of (128 word) records and assigns it the name specified. End of file (EOF) marks are written in each record.

Serial PRINT# Statement

PRINT # file no. ; data list [, END]

Stores data in the specific file, after the last item read or printed. End of record (EOR) marks are used to fill the rest of the last record containing data. When the optional END parameter is used, however, and end of file (EOF) mark is placed immediately after the last data item; then the rest of the record is filled with EORs.

D

Random PRINT# Statement

PRINT #file no., record no. [;data list[, END]]

PRINT #file no., record no.;END

Prints specified data items in the file number indicated, starting at the beginning of the record number specified. The optional END parameter places an EOF mark after the last data item printed in the last record. When END is not used, an EOR mark is placed after the last item printed. In either case, the rest of the record is filled with EORs.

Logical PRINT# Statement

PRINT #file no., record no., word pointer [;data list[, END]]

Stores data items in a specified record of a file, beginning at the specified word. The file number and record number are the same as with random file access. The word pointer can be an integer expression from 1 thru 129, and specifies where the first word of data is to be printed. The optioal END parameter places an EOF after the last data item printed. When END is not used, the remainder of the record is left unchanged.

PRINT LABEL Statement

PRINT LABEL, drive no., "text" or string variable

This statement assigns an alphanumeric label to the disk in the specified drive. The label can be up to 224 characters long.

Serial READ# Statement

READ #file no. [;data list]

Reads data from the specified file, starting after the last item printed or read.

Random READ# Statement

READ #file no., record no. [;data list]

Reads data from a specified file, starting at a specified record. When the data list is omitted, this statement repositions the file pointer to the beginning of the specified record in the file.

Logical READ# Statement

READ #file no. , record no. , word pointer [, data list]

Reads numbers and strings into variables from a specified record, starting from a specified word.

READ LABEL Statement

READ LABEL , drive no. , string variable

Reads the label on the disk and stores it in the specified string or substring. If the disk doesn't have an assigned label, the null string is returned.

REC Function

REC file no.

Returns the current position of the record pointer within the specified file.

REDIM Statement

REDIM array list

Changes the matrix working size of the specified array(s) to new specified boundaries. The array subscripts must be equal to or larger than the original array size. (The working size of a matrix is the same as the physical array size unless it is redimensioned.) When subscripts are not used, the working size is redimensioned to the array's original size.

RESAVE Statement

RESAVE file name [, 1st line number[, 2nd line number]]

Stores a new program, or the lines indicated by the line numbers, on the disk using a previous file name. The same line number rules apply as for the SAVE statement.

SAVE Statement

SAVE file name [; 1st line number[; 2nd line number]]

Stores an entire program, or the lines between and including the specified line numbers, into the file named.

SAVE KEY Statement

SAVE KEY file name

Stores all present special function key definitions in the named file.

SAVE MEM Statement

SAVE MEM file name

Stores the entire read/write memory in the specified file.

SIZE Function

SIZE file no.

Returns the size, in records, of a specified file.

SLEN Function

SLEN file no.

Checks for strings in the specified file: if the next item is a string variable, its length, in characters, is returned. −1 is returned if the next item is not a string.

STD Function

STD

Returns the standard printer select code currently specified by the STDPR statement.

D

TYP Function

TYP file no.

Identifies the type of the next data item to be accessed in a specified file. TYP returns one of these codes –

Type Code	Meaning
0	Item not printed via the 9831A.
1	Next item is a full-precision number.
2	Next item is a string variable contained in one record.
2.1	First part of a multi-record string.
2.2	Intermediate part of a multi-record string.
2.3	Last part of a multi-record string.
3	Next item is an EOF or the physical end of file.
4	Next item is an EOR or the physical end of record.
5	Next item is a split-precision number.
6	Next item is an integer-precision number.

UCASE Statement

UCASE string variable

Converts all characters in the specified string or substring to uppercase.

UNIT Statement

UNIT drive no. [, select code]

Specifies the disk drive number (0 thru 3) and optionally, the drive select code (8 thru 15), to be used by subsequent disk operations. UNIT changes the drive number (0) and the select code (8) automatically specified for disk operations when the 9831A is switched on.

D

UPOS Function

`UPOS` (string variable, "text" or string variable)

Returns the position of the first character of the second string within the first string. The strngs are temporarily converted to uppercase before being compared.

WCTL Statement

`WCTL` select code, expression

The WCTL (write control) statement outputs a binary number for controlling various functions on the specified interface card.

WRD Function

`WRD` file no.

Returns the current position of the word pointer for the specified file. The value can be from 1 thru 129; 1 indicates the first word of the record, while 129 indicates that the next item to be printed will go into the first word of the next record.

D

Error Messages

Disk Drive (hardware) Errors

78	I/O interrupt: for example, an interface card is plugged in while power is switched on.
79	All disk drives not switched on.
80	Disk drive door open.
81	Disk not installed or specified drive number not set.
82	Write-protected disk.
83	Disk drive record-header error.
84	Disk track not found.
85	Disk data checkword error.
86	Disk drive hardware failure. Press RESET to regain system control.
87	Verify data error: occurs during auto-verify routine. Try to reprint the data.

Flexible Disk ROM Errors

88	Miscellaneous disk ROM syntax error: for example, storing an incorrect IF END# statement.
89	Incorrect disk drive number or select code. Also, incorrect record pointer or word pointer.
90	Incorrect disk file name or file not found.
91	Available disk file space exceeded; also directory or availability table is full.
92	File name already exists on drive.
93	EOF (end of file) mark reached or physical end of file encountered.
94	Disk file format error: for example, a multirecord string not intact.

D

Subject Index

A

B

C

D

E

F

G

H

I

K

L

M

N

O

P

R

S

T

U

V

W